

Rethinking CPU Design

in the early 21st century

copyright © Gord Deinstadt, 2010

Table of Contents

Introduction.....	2
Part 1: CPU Architecture.....	10
Part 2: OVPU Architecture.....	25
Part 3: Memory Management.....	33
Part 4: The Instruction Cycle.....	42
Part 5: Context Handling.....	47
Part 6: Instruction Reference.....	53
Part 7: Extended Instructions (64-bit only).....	72
Part 8: OVPU Command Reference.....	76
Part 9: Implementation.....	85

Introduction

For decades now the standard textbook for computer design has been Computer Architecture: A Quantitative Approach¹, now in its fourth edition. This textbook is not intended to supplant that textbook but to supplement it with an alternative view. My emphasis is on things that are overlooked or under-explained there. In particular, Quantitative Approach seems to assume in a rather sunny way that whatever is being sold must be well-designed. I don't think that is the case. CPU design has been at an impasse for some years now. Multi-core is not a scalable solution, and it is not what customers really want, it is simply what manufacturers have to sell. I argue that real CPU design – original design, not just iterations of planned obsolescence – is almost absent from the market today.

This book describes, as an example of original CPU design, the operation of and thinking behind the Gordotron-32 and Gordotron-64 computers.

I am opposed to patents on practical, moral, and religious grounds. I wish that anyone should be free to manufacture and sell these computers and computers derived from them. The designs described here are freely available to anyone. You may fabricate, sell, distribute, and use these designs for any purpose without any obligation to me. However, they come without any warranty. In particular I do not warrant that they do not transgress against any particular patents in any particular jurisdiction. I have included discussion of old prior art where appropriate so you can see what things are probably in the public domain. I also warrant that this design is entirely original to me (except where I have discussed prior art) and that I have never applied for patents related to these designs.

Anyone is welcome to start or contribute to a project to implement these designs on opencores.org. I would do it myself but I don't have the resources to generate the design files.

In addition, such code fragments as are included in this book may be copied, distributed, derived from, or otherwise used for any purpose without restriction.

¹ Computer Architecture: A Quantitative Approach, 4th Edition. Hennesey, John L. and Patterson, David A. Elsevier Digital Press, 2007. ISBN 978-0-12-370490-0.

Background

Principle: *Engineering takes place in the real world. Sometimes the real world sucks.*

Hardly anyone is a CPU designer anymore. Many people are computer system designers, or chip designers, but hardly anyone designs their own CPU nowadays.

It was not always thus. Back in the early 1970s, it was feasible to run up your own CPU design. Everybody had access to the technology, which consisted of stuffing small packages of logic gates into a printed circuit board. At that time integrated circuits were limited to Small-Scale Integration (SSI, tens of gates per package) and Medium-Scale Integration (MSI, hundreds of gates per package). So, anyone could do what the computer manufacturers did. Consequently it was not uncommon for products like telephone switches, industrial robots, and test and measuring equipment to include custom-designed CPUs. That is when I first had the opportunity to design a CPU.

Times have changed. Nowadays there is tremendous pressure to use an off-the-shelf CPU, not only because it is cheaper but because off-the-shelf CPUs come with operating system and compiler and development system support, with testing and QA support, and with a cadre of engineers who are familiar with them. Nowadays, even if you are designing a custom chip, you incorporate an off-the-shelf CPU into it.

Besides that, outside of embedded applications the system designer usually does not have a choice of CPU. If the CPU is exposed – and sometimes even when it is not – the CPU choice is determined by marketing or by some other non-technical manager. And you know what they always want; they want you to build your real-time system on Windows and i86. They are risk-averse, and because they are technically ignorant they think that Windows/i86 must be the lowest-risk platform. Windows/i86 is the best-selling platform (in their eyes, since they don't know anything about the embedded market) so it must be the best. They simply won't believe you when you tell them otherwise.

I hear you ask, what does marketing have to do with it? Here is something every engineer needs to know. The difference between sales and marketing is that sales is fulfilling demand while marketing is creating demand. And how do you create demand? One way is to come up with new products. So product engineering is – ta da! – a subset of marketing. I am not kidding. In large corporations it is the marketing department that determines the specifications for products, sometimes in detail.

So in most cases, apart from embedded applications, all the system designer gets to choose is which model of CPU, what kind of RAM and how much, and so on. These are tradeoffs and they require engineering skill of the sort taught by Quantitative Approach, but they do not require creativity or originality. This is engineering with all the *fun* drained out of it.

For embedded applications, by contrast, the CPU is hidden from the user and therefore usually marketing doesn't care about it. That makes engineering the product much more fun.

Legacy is also a trap for manufacturers. Intel has repeatedly tried to dump the i86 architecture. The 8080 was deliberately binary-compatible with the 8008, but the 8086 (the first Intel 16-bit MPU) was only source-compatible with the earlier machines.

Later, when 16 bits was not enough, Intel brought out the iAPX-32, a deliberately incompatible machine. Indeed they were so determined not to be locked in that they refused to expose the native instruction set of the iAPX-32. You had to use the Intel-supplied Ada compiler. This seemed like a good idea at the time because Ada was being heavily promoted by the US Department of Defense as the new universal language. Unfortunately for Intel Ada did not catch on outside the military sphere. And,

customers complained that the iAPX-32 was slow. Meanwhile Intel had deliberately limited the 8086 family to the 80286, which was still a 16-bit architecture but with a level of indirection added to segments to increase the address space. When the iAPX-32 was finally canned the 8086 team was allowed to come out with the 80386, the first true 32-bit member of the family. However for backwards compatibility it also had to emulate the awkward 80286 segmented mode. Then Microsoft screwed Intel around by refusing for years to move to 32-bit code, and then by employing a mixture of code in 80386 mode and code in 80286 mode. As a result Intel had to preserve this mess all the way through. It is still in the current chips, but nowadays Intel chips hold a lot more transistors, so the transistor count for 80286 mode is no longer a big deal. Nonetheless, it is wasted hardware that users are obliged to pay for.

Intel tried again at the transition to 64 bits. Once again they came out with an incompatible chip (the Itanium) and blocked the 8086 (now re-branded i86) team from introducing a 64-bit version. This time they had an agreement with HP to co-manufacture, and with Microsoft to port Windows. Again the new architecture has been less than a stellar success in the marketplace. It has not been withdrawn, perhaps because Intel is bound by contracts. However, Intel is not updating it as often as they need to. And once again Intel was forced to introduce a 64-bit i86 chip, this time because AMD introduced their own and forced Intel's hand.

As a result real innovation has come primarily from the embedded market. The prime example is cell phones. The number of cell phones in use has quietly grown to 5.5 billion worldwide, for a world population of 6.7 billion. 500 million more phones join the ranks every year, so in 2012 we will hit one cell phone for everybody in the world. (I will be the only person without one, still stubbornly refusing to make myself available to all and sundry every hour of the day and night.) The cell phone market far exceeds the desktop computer market, or indeed the computer market generally, in number of units shipped and in number of units in use. Furthermore, it generates more money, because each of those phones runs up monthly charges. So this is where the large and growing market lies, not on the desktop. And it is a much more open market from an engineering standpoint; the major player with the least to offer in this market is Intel. The majority of pocket devices use CPUs from the ARM family.

Now the pocket and desktop markets are colliding. This has opened up a new window of opportunity for real hardware innovation. Furthermore, this time around manufacturers are not foolishly locking themselves into a single architecture. They are storing and/or distributing apps in a form that is hardware-independent. (Not that they generally want to pass any of that freedom onto the consumer.) That makes it possible for manufacturers to offer devices that are based on non-traditional CPUs.

On the server and data center side, more and more servers are running Linux. So far Linux has mainly cannibalized the Unix market, but with continued support from IBM and others it is starting to make inroads into other territory. This is good for CPU designers because anybody can port Linux to any CPU. There is also an open-source development environment, Eclipse, so it is possible to equip a new CPU with the necessary tools. By contrast in the desktop market most applications use Microsoft's development tools, which are locked to Intel CPUs.

Currently Intel is concentrating on multicore designs. The server and data center are the places where multicore architectures have real value. When you are executing thousands of jobs concurrently you can always employ more cores. This is not true for the desktop and laptop markets. In those market anything more than 3 or at most 4 cores will just be wasted. Hence the current fad for multicore chips does not reflect the market as a whole. It reflects a failure of imagination on the part of CPU designers and business models on the part of CPU builders. They produce multicore chips because they don't know what else to do.

Some people think that someday all software will be converted so that it can use as many cores as are available. This is a vain hope. There are many common algorithms, dictated by standards and usage, that cannot be executed in parallel. For example, many communications and data formatting protocols include an MD5 hash. MD5 cannot be executed in parallel; each step requires the result from the previous step. Even if we remove MD5 from all the standards, there is still a vast array of existing files containing MD5 hashes. Whenever we go back and look at that data we have to verify the MD5 hash using sequential processing.

For tasks that can be performed in parallel there is already a set of specialized processors that do it very well. They are called GPUs (Graphics Processing Units). This is an area where there is still interesting work going on, even in the mainstream. Most importantly they are working on applying their technologies to new venues, not just graphics but physics and other calculations.

Economics of Chip Manufacture

“In a period of significant competition, price tends to track cost closely, although microprocessor vendors probably rarely sell at a loss.” - Quantitative Approach, section 1.6.

Wow. AMD lost money every quarter but one for 10 years. Let's talk about the economics of chip manufacture.

The microchip manufacturing business illustrates market failure in an industry dominated by sunk costs. In English what that means is that the chip market does not work the way markets are supposed to work. The problem is that most of the money is not spent making chips, most of it is spent building the factories (for some strange reason called “fabs”). These days it costs a billion and a half dollars for a single fab. That is a staggeringly huge up-front investment for private industry. As a result, when a chip is manufactured, it may only cost \$5 to make (the “marginal cost”) but in order to pay the mortgage on the fab they have to sell it for \$50.

Now demand drops or supply increases and the market won't pay \$50 that chip anymore. Maybe they will only pay \$40. In a normal business, dominated by marginal cost, the chip manufacturer would lay off employees and reduce production, thereby bringing supply and demand back into balance. But in the chip business this doesn't work. Reducing production won't pay the mortgage! Neither will continuing to produce, but at least if you produce and sell you are still making some money on each chip (in this case \$35), so you don't sink further into debt as quickly as you would if you reduced production. The result is that all manufacturers continue to produce at full speed, and the price sinks even lower. The supply and demand feedback loop does not function.

The reason should be obvious to anyone who has studied control theory. The fab cost is sunk cost, that is to say it has already been spent. There is no way to get it back. (You could sell the entire fab but then you'd be out of business. And you'd get a lousy price, because chip prices are low.) So for a control system such as a market to work, it would have to send a signal into the past and alter history. Since it cannot do that, it cannot possibly provide stable negative feedback.

By the way, this is the same problem farmers have. They have already sunk the cost of buying the farm and growing the crop before they find out what their yield will be and what price they can get. If they are growing something storable (like grain) they can at least average across years. But if they are growing something that is not storable (like vegetables or livestock) they can instead average money via crop insurance. Chip manufacturers don't have either of these options. Chips become obsolete rapidly, so they are not storable. And no one is stupid enough to sell business insurance to chip manufacturers.

So the result is that, every time there is a price war, one manufacturer goes out of business. This

reduces output temporarily (until the fab is sold) and restores the balance of supply and demand. If it was not for governments pushing new companies into the chip business we would long ago have been reduced to just one chip manufacturer. Or two: once a market gets cozy enough the players usually stop competing. AMD versus Intel is very unusual in that respect.

There is one other solution, and AMD finally adopted it. You can split off the fab from the rest of the business as a contract chip foundry. That way, if your CPUs are not in demand, the fab can manufacture some other type of chip for another customer. It's not a guarantee of survival but it is an improvement. Intel is the only remaining major manufacturer with an in-house fab rather than a foundry business.

Failure of the Standard Approach

For many years Intel copied Seymour Cray's innovations (as also did AMD). The Pentium series was not about original design but about applying supercomputer techniques to an existing family of processors. This was feasible because the chip manufacturing technology kept allowing them to pack more and more transistors on a chip. The single biggest innovation was lengthening the pipeline.

Intel's marketing and its engineering fed upon each other. Long pipelines enable, but also require, high clock speed. For a long time Intel managed to totally baffle the average user and business manager by equating clock speed to execution speed. Intel chips have always had a comparatively high clock speed for their throughput, so their marketing department took advantage of that. Essentially the Intel line was "Look at the size of my clock! Just look at it!"

Pipelining seems like a simple enough thing when the pipeline is only two or three stages long. However as the pipeline length goes up pipelining becomes very expensive. An N-stage pipeline does not cost N times the number of gates, in fact it may approach N^2 .

Consider instruction fetch and decode. The more deeply you prefetch and predecode instructions the more likely you are to run into a branch, so the more parallel fetch and decode units you need. Nowadays it is common to use five of them. But that's not all! All those parallel fetch and decode units have to be fed somehow, so you need proportionately more bandwidth within the instruction cache. All this to fetch and decode five sequences of instructions even though not more than one sequence is going to be executed.

Then came the P6. This was supposed to be just another turn of the crank, an even longer pipeline than the 26-stage P5 pipeline, capable of clocking at a higher speed. But the P6 failed because it generated too much heat. Using air-cooled packaging it was impossible to run it at its design clock speed. But at lower speeds, within the range of the P5, the P6 was slower because of its long pipeline. Pipelines increase the clock rate but they also introduce more stages of logic, so if you can't run at the higher clock rate you lose.

Intel asked their customers if they would adopt water cooling, but the answer was a resounding no. Finally Intel was forced to withdraw the P6 from the market. It was a disaster, and the company was in deep trouble. AMD was making Opterons² that ran at lower clock speeds but delivered more punch than a P5. AMD could speed up; Intel could not.

One of the solutions that Intel took was illegal arrangements with customers. Sometimes it was just a threat; if you sell any AMD chips we will cut off your supply of Intel chips. (This is illegal restraint of trade.) In the case of Dell it was kickbacks that might not have been illegal in themselves, but the money was paid into a secret slush fund and used by Dell management to mislead investors. This is all

² This book was written on a dual-core Opteron.

documented; prosecutions and convictions have taken place.³ Nonetheless it was worth it for Intel, because it helped buy enough time to save the company.

Another thing that saved them was that AMD simply could not ramp up production quickly. They didn't have the capacity, and you can't get chip manufacturing capacity quickly. It takes years from when you place your orders before the production line is in place. AMD did apparently contract out some manufacturing to IBM, but the market is such a big one that IBM's capacity did not help much.

But the main thing that saved Intel was a brain transplant. There was a small company in Israel, only about 40 people, who had created a chip that was i86 compatible but used lower clock speeds and shorter pipelines and was more energy-efficient. Intel bought the company, and it has now grown much larger and become their chief design center. Most of the latest Intel designs come from there.

Reducing Entry Cost

One of the biggest problems for innovative designs is high entry cost. But there are things a designer can do to reduce the cost of entry.

A good many low-end MCUs are made on obsolete production lines. When a fab becomes obsolete, perhaps because the industry moves to larger wafers, the equipment is not just scrapped. It is sold to niche producers who move it to low-wage countries and set up trailing-edge manufacturing facilities. This means that, as long as you can make your design work with a rather ancient technology, you can get it made much more cheaply. The masks are simpler and cheaper than more modern masks, and the cost of the fab is much lower so the economics is much more attractive.

However, since graduating to 30 cm diameter wafers the industry has not moved on. There is very little interest in moving to still larger wafers. Furthermore, the lithography equipment (known as “steppers”) has also stalled. The next generation, which will use Extreme Ultra-Violet (EUV) has proven very difficult to develop and is far behind schedule. Instead chip makers have pushed the previous generation far beyond its original capabilities using phase-compensated masks, double-patterning, and liquid immersion.

Because the previous generation of steppers has had its life extended, there are no obsolete steppers to equip low-end chip manufacturers. However, the methods mentioned above to extend the capabilities of the steppers add a lot of costs. You can still save money by making a chip on the current equipment with relaxed 110 or 130 nanometer rules, because the masks are simpler (no phase compensation) and/or fewer (no double-patterning) and the production process is simpler (no liquid immersion).

Using wider lines means that the circuit takes more area, but for MCUs⁴ the overall chip size is often dictated by the bonding pads anyway. If that is the case, you get the same number of chips per wafer regardless of the line width.

Of course when you use bigger transistors your gates are inherently slower. However propagation delay can actually decrease because you can make wider lines thicker, lowering resistance per unit length as the square of line width. Therefore overall speed may not suffer. And larger transistors run at higher voltages, which in many cases means lower leakage in the off state, so power consumption does not suffer as much as you might expect.

Hence, for a chip with a modest transistor count, using relaxed rules means you pay less for each chip

3 Of course, the world being what it is, none of the individuals who conspired to break the law has had to serve time in jail or even personally pay a fine.

4 Micro Computing Units, the one-chip computers that are usually used in embedded applications. As opposed to Micro-Processor Units (MPUs), which require a board full of other chips to make a working computer.

while getting nearly comparable performance. At the same time it means you have a wider choice of manufacturers, because not all of them can manage small line widths. It is therefore still desirable to reduce transistor count well below the state-of-the-art.

Finally, it is desirable to lower the transistor count in order to lower the cost per chip. It is true that actually making more transistors may not cost any more, but you can't make them until you have designed them and debugged them and documented them. That's extra investment⁵ up front. Then, when you have made them, you also have to test them. Testing costs can easily exceed actual fabrication costs. So as always *Keep It Simple, Stupid*.

Outstanding Issues in CPU Design

You can look at the failure of deep pipelining in two ways: either as a painful loss or as an opportunity. Any CPU designer with ambition and imagination will see it as an opportunity. Combined with the smartphone invasion, we have a chance to free ourselves from the prison of instruction set compatibility. There is, for the first time in many years, an opportunity for real innovation.

How can we get rid of the burden of long instruction decode pipelines? By using instructions that don't need to be decoded. This is the principle of Very Long Instruction Word (VLIW) architectures. But the name reveals the drawback; the instructions have to be very long, often more than 100 bits. That length increases the instruction fetch bandwidth, giving back some of the gains made by eliminating instruction decode. And the long instructions increase program size, which necessitates both a larger main memory and a larger instruction cache.

The answer to this problem is to employ finesse in the design of the instruction set. To me an instruction set is really an industrial design. It is a tool crafted to fit the needs of the programmer. It rarely does anything new, but it may do the same thing in a more convenient and/or elegant way.

What an instruction set is *not* is an architecture. Intel marketroids introduced the phrase “Instruction Set Architecture”. This is Orwellian [newspeak](#). There is no single instruction set for all Intel processors, not even for all those that belong to the i86 family. Even if there was this would not constitute an architecture. The instruction set is only a small part of the machine architecture. A lot of the architecture is hidden under the covers, or visible only to hardware geeks. Even the portion that is visible to programmers includes device registers, privilege modes, memory management modes, sleep/wake modes, interrupt handling, and many other features.

When Intel introduced the phrase “Instruction Set Architecture” they were trying to confuse people into thinking there was more planning to the i86 series than there really was. It is exactly the same thing IBM did in the 1970s when they introduced the phrase “System Network Architecture” to conceal the fact that they had implemented a bunch of incompatible communications protocols with no strategy or overall architecture at all. That's why, when I hear “*Something Something Architecture*” spoken by a marketroid, I reach for my gun.

There are two additional issues that current architectures do not address, but that must be addressed.

1. The imbalance between CPU speed and main memory speed. The vogue for multi-core MPUs has only made this worse. Symmetric Multi-Processing (SMP) is a simple hack to convert a single-processor operating system into a multi-processor one. But the cost is too high. SMP simply is not scalable. In the long run we need a different operating system architecture, one built from the ground up for the cloud. I will address that issue elsewhere. In the short run we can at least rectify the

⁵ Known as “NRE”, which stands for non-recurring engineering costs.

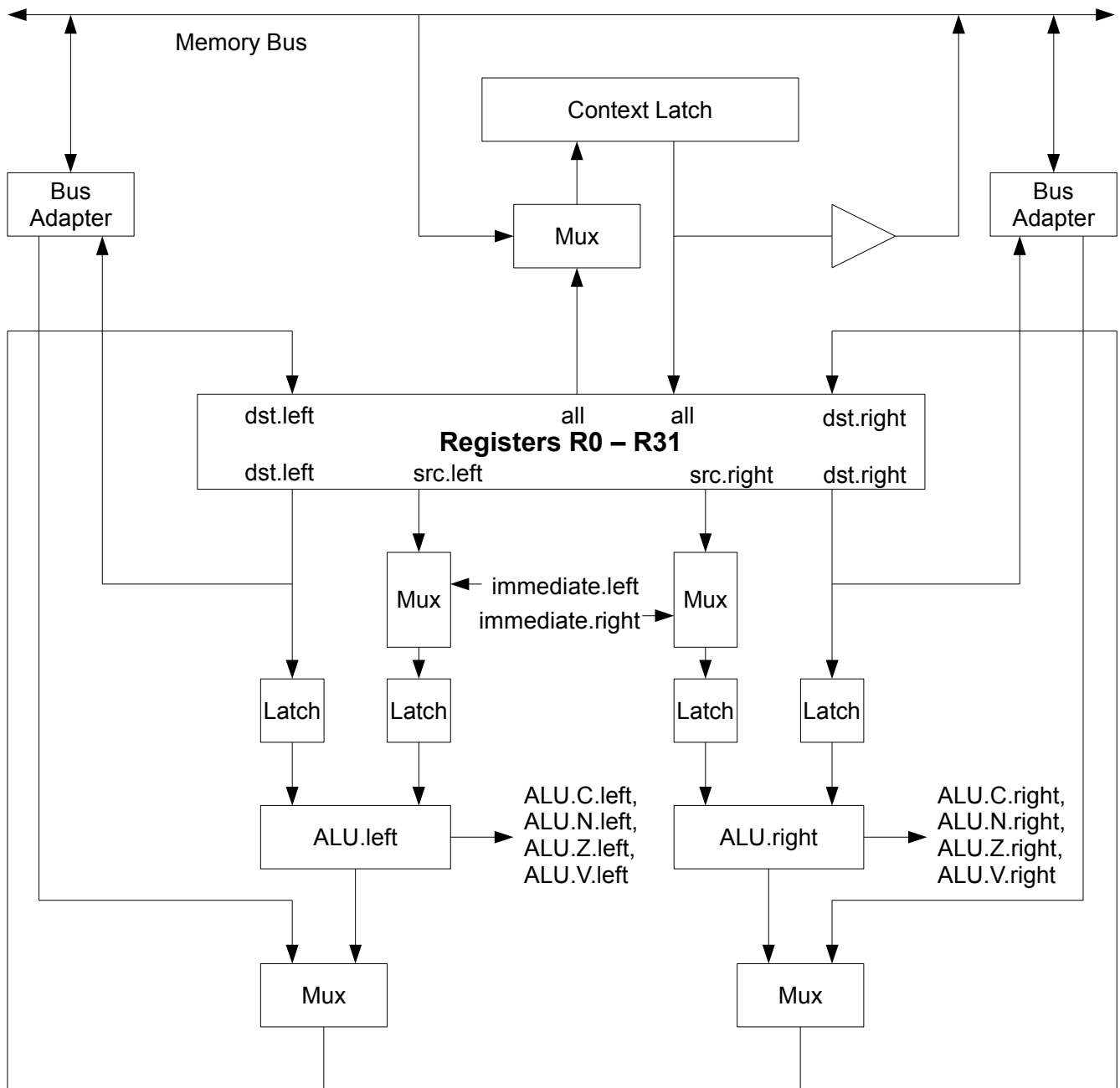
bandwidth imbalance between CPU and memory, even if we cannot fix the latency problem.

So we increase the bus width. For a machine with 32-bit words, and 32 general registers, I propose to increase the bus width to 1024 bits. It turns out this has a whole lot of consequences, mostly good ones. For example, it means that we don't have to do an instruction fetch very often; when we fetch we fetch a whole block of instructions. This in turn means our instruction format is not as tightly constrained as it usually is in RISC machines. And, it means that, if we supply an appropriate engine, we can operate on whole objects or strings in a single memory cycle. More work gets done, less time and energy is wasted moving data around a wee piece at a time.

2. Entangled machine states. These entanglements are sometimes explicit, such as status flags that mix the results of multiple instructions. But they are also hidden, such as execution times that depend upon the global cache state. Either way entangled machine states are bad news. Entangled status flags invite bugs that are all but impossible to find under test. Entangled execution times make it impossible to write real-time code that is correct by design.

We no longer tolerate such lack of modularity in programming languages. We should not tolerate it in hardware design either.

Part 1: CPU Architecture



Basics

Principle: *Don't hide what you are doing from the programmer.*

The Gordotron-32 codes operations in a way that is quite compact yet only takes a few gate delays to decode. This is done using only a few bits to select a general type of operation and then additional flag bits to modify the basic instruction into a variety of instance operations.

A complete operation with two operands is coded in only 16 bits. Therefore, each 32-bit instruction word holds two instructions. Each instruction is associated with its own dedicated processing unit, one for the left instruction and one for the right instruction. The result is a superscalar processor with nearly the simplicity of VLIW but with high instruction density.

Registers

There are 32 32-bit word registers, identified as R0 through R31. Any of R0 through R29 can be used for any purpose. R30 is dedicated to the status register and is also known as the SR. R31 is dedicated to the program counter and is also known as the PC. Branches are performed by naming the PC as the destination of an instruction. Reading the PC returns the address of the word immediately following the instruction.

Each of the processing units has its own ports for accessing registers R0 through R30. Hence most instructions can be performed by either the left or right unit equally. What happens when the two instructions write to the same register? The resulting content of the destination is the bitwise AND of the result data from the two instructions. This is convenient for extracting a bit field or a byte.

```
shift.right.0 ALIGN, R0; copy MASK, R0
```

Note that the implied AND operation does not take any extra time – it is a side-effect of the multiport register logic.

The left-hand processor can both read from and write to R31 (the PC), but the right-hand processor can only read from it, not write to it. Branches, therefore, must be executed by the left unit and no conflict can occur between the two units.

The status register (SR) and the program counter (PC) each has an extra read port and write port known collectively as the “back door”. These are provided so that these registers can update when they are *not* explicitly named as destination by either instruction. However the back door has a lower priority for writing than the front door. In other words, when either instruction names either the SR or PC as its destination, the normal automatic update of that register is suppressed.

Advantages of explicit parallelism

There are advantages to having two instructions operating in parallel. For example, modern CPUs don't normally include a register swap operation because this would require an extra write channel. But with this design, we can use the two parallel processing units to perform the swap in a single cycle. And it doesn't require a dedicated instruction, either. Instead we tell the processing units to copy opposite directions simultaneously.

```
copy R0, R1; copy R1, R0 // swap R0 and R1
```

This works because each unit latches its operands early in the cycle and writes its output late in the cycle.

Regular architectures require a dedicated instruction and extra hardware or cycles to perform subroutine

calls.⁶ The Gordotron-32 just uses both processing units together.

```
jump SUB1; copy PC, R29 // call SUB1, saving return address in R29
```

Here is the maddening thing. Most modern CPUs are at least dual-issue. So you could do these things with them, too. *But they don't let you.* Instead they spend a bazillion gates maintaining the illusion that they are single-issue machines. That is just stupid.

Explicit parallelism also makes it possible to operate on 64-bit operands using both execution units in tandem. In this mode the left-hand unit is slaved to the right-hand unit; the operation is determined by the right-hand unit, modified by a flag bit from the left-hand unit. Both units supply operands; since each operand is only 32 bits, we need both sets to build out 64 bits. The whole operation takes a single cycle, so it is fair to say this machine supports both 32 and 64 bit native operations.

Note: When you supply the assembler with only one instruction for a word, it places the supplied instruction on the left side and inserts a `nil` on the right side. For example,

```
copy R0, R1
```

is transformed to

```
copy R0, R1 ; nil
```

The upshot is that the solo instruction affects only the result flags for the left unit. The result flags for the right unit are unchanged.

Restrictions

Some instructions (if, tandem, jumps and branches) are constrained to only operate in the right or the left slot. A single co-processor instruction occupies both slots but can use two sources and two destinations.

Gordotron-64

The 64-bit version cannot pack four instructions into 64 bits. 32 registers are not enough to keep four execution units busy. Instead the register count has to expand to 64. (Besides, this also allows the source field to designate one of 64 bits in a word.) The result is that each instruction expands to 18 bits and only three will fit in a 64-bit word, with 10 bits left over.

There is a silver lining to this cloud. Six of the left over bits can specify a second source operand for one of the execution units. Thus in the Gordotron-64 one of the three instructions executed in each cycle can be a three-operand instruction. This is plenty to meet normal three-operand needs.

3 of the remaining 4 bits are used to enlarge the op code fields for the three instructions. This allows plenty of room for additional op codes, such as floating-point operations. The Gordotron-32 does not really have enough registers or big enough registers to keep up with hardware floating-point, but the Gordotron-64 does.

The remaining bit of the instruction word is not used and must be zero.

A tandem instruction in the left position slaves it to the center processor. A tandem instruction in the center position slaves it (perhaps in turn) to the right processor.

⁶ If a machine exposes a branch delay slot you can save the return address in the delay slot. But an exposed branch delay slot *is* explicit parallelism.

Conditional Operations

Principle: *Reuse functionality.*

Principle: *Eliminate tangled states.*

It is possible to fully specify a conditional operation in one cycle using explicit parallelism. The Gordotron-32 uses the right instruction in the pair to test a condition and impose its result onto the left instruction in the pair. If the right-hand unit determines that the condition is not met it inhibits the write pulse for the left-hand unit, disabling that instruction. For example,

```
loop:
...                               // do stuff
decrement R17; ...                // decrement counter and do something else
branch loop; if.gt.left           // iterate until counter reaches zero
```

The left-hand instruction of the pair *may* be a branch but it does not have to be. For instance, assuming that no hardware multiply is available, the following code performs a signed 32-bit by 32-bit multiply yielding a 64-bit result.

```
// Enter with multiplier in R0, multiplicand in R1
// Product is built in R2 and grows into R2,R3
clear R2:R3                      // tandem clear
add R1, R2; if.1 0, R0           // carry out to SR.c.left
shift.right.c 1, R2:R3          // 65-bit shift out LSB, shift in SR.c.left
add R1, R2 ; if.1 1, R0
shift.right.c 1, R2:R3
...
shift.right.c 1, R2:R3
add R1, R2 ; if.1 31, R0
```

The multiply takes 64 cycles, which is pretty good for a machine with no multiply-step instruction.

A conditional can test any arbitrary bit of any register for the value 0 or 1. Of course, it has to also be able to test the usual set of conditions: equal to zero, greater than zero, and so on. The condition code bits in the status register are defined to make it possible to test any standard condition by looking at one bit. To accomplish this some conditions are pre-decoded and appear as result flags.

The lower half of the status register (SR) contains two sets of result flags, one for the left processing unit and one for the right processing unit. After a 64-bit operation the result flags for the left unit reflect the result for the entire 64-bit operation.

<i>Left Bit</i>	<i>Right Bit</i>	<i>Name</i>	<i>Access</i>	<i>Description</i>
15	7	SR.z.* (SR.eq.*)	read-write	result was zero (src equals dst)
14	6	SR.n.*		result was negative
13	5	SR.c.* (SR.lo.*)		carry out (unsigned src < dst)
12	4	SR.v.*		signed overflow
11	3	SR.lt.*	SR.n.* ^ SR.v.*	signed src < dst
10	2	SR.le.*	SR.z.* SR.lt.*	signed src <= dst
9	1	SR.ls.*	SR.c.* SR.z.*	unsigned src <= dst

* Bit names end in .left for the left unit and .right for the right unit.

Many CPUs try to preserve result flags across instructions where possible. For example during a bitwise OR operation they preserve the overflow flag because it cannot convey any useful information about the OR operation. The idea is that, since preserving and later testing status is difficult, status bits should be preserved when possible. This is not necessary for the Gordotron-32, because preserving and testing status flags in software is easy. Furthermore, preserving flags in hardware results in an entangled machine state; some flags represents the previous instruction, others an instruction some cycles earlier. This encourages software bugs. For that reason the Gordotron-32 deliberately does not preserve any result flag across any instruction, except for two special cases:

1. Any instruction that performs a branch preserves the result flags for that unit. This makes it easier to call into or return from a subroutine with a precomputed status. This includes both instructions that explicitly specify the PC as the destination and three (branch, next, and trap) that have implied destinations.
2. A conditional instruction (which is always a right-hand instruction) preserves the result flags of the right execution unit. In addition, if the condition is not met, it preserves the flags corresponding to the left execution unit. This makes it possible to stack conditional operations, for example to execute a three-way branch corresponding to less than, equal, or greater than.

When you need to keep the status for future use you have to copy the SR to another register or memory location. The conditional test instruction operates just as efficiently on other registers containing saved status information as it does on the SR.

```
copy SR, R4; ...           // save status from previous instruction
...                       // do other stuff
... ; if.lt.right R4       // test saved status for right-hand signed less than
```

Other than the exceptions noted above, each instruction in a pair updates all of the result bits for its side of the instruction.

- The Z bit is set if and only if the result was all zeros.
- The N bit is set if and only if the MSB of the result was 1.
- The C bit is loaded with the carry out from arithmetic operations or the last bit shifted out from shift operations, but is otherwise cleared.
- The V bit is set by arithmetic operations if the operation generated an overflow, considering the operands as signed. In other words it is set if two positive operands yielded a negative result or two negative operands yielded a positive result. For most other instructions the V bit is cleared. However there are some other instructions that can encounter data that is not valid, for example the encode instruction may encounter an all-zero source operand, for which there is no possible priority encoding. In such cases the V bit is set, in addition to whatever the other result bits do.

There is a bit in the status register, SR.nil, that always reads as zero. This can be tested for 1 to yield a skip operation (do not execute left or right hand) or for 0 to yield a nil operation (right hand does nothing, left hand executes as usual).

<i>Bit</i>	<i>Name</i>	<i>Access</i>	<i>Description</i>
16	SR.nil	read-only	always zero

Debuggers need to be able to insert a replacement instruction that will cause a trap and yet yield the same status after execution as at the beginning of execution. Debuggers should use the following instruction pair to achieve this:

```
trap BREAK ; nil
```

Semaphores

It is sometimes necessary to test and conditionally update a memory location in a single indivisible (atomic) operation. This is used for a semaphore, which is an object employed by multiple processes to co-ordinate access to a shared resource. There is a variant of the store instruction that implements a semaphore using a special read-modify-write bus cycle.

Gordotron-64

On the Gordotron-64 either the center instruction slot or the right instruction slot may contain a conditional instruction. If it is in the right instruction slot it controls both the left and center slots. However if it is in the center slot it only controls the left slot.

Instruction Types

Principle: *Orthogonality rules...*

Instructions operate between a source and a destination, usually altering the destination. The source can be any register (R0-R31) or an immediate value. All instructions except co-processor instructions use the following format, although some of them reinterpret the destination field. Remember that there are two of these instructions in a word of code, a left one occupying bits 16 through 31 and a right one occupying bits 0 through 15.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op Code + Flags					I	Source Field					Destination Field				

Destination field

The destination field selects one of three things:

For implied-operand instructions, it acts as an auxiliary op code.

For put and get instructions, it selects an object buffer. The destination can be any of BF0 through BF31.

For all other instructions, it selects one of the general registers. For these instructions the destination can be any of R0 through R31, however only the left execution unit can write to R31.

Source field + I

There are three different ways to specify the source operand. In *register source* format the source operand is the content of any of R0 through R31. These instructions are signified by a 0 in the I bit. For *short immediate source* format the source operand is a small positive value stored in the instruction itself. For *long immediate source* format the source is a 32-bit value which is appended to the instruction. Both short and long immediate instructions have a 1 in the I bit.

Register source format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op Code + Flags					0	Source Register					Destination Field				

Short immediate source format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op Code + Flags					1	Source Operand					Destination Field				

Long immediate source format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op Code + Flags					1	1	1	1	1	1	Destination Field				

All instructions can use the register and short immediate source formats, but not all instructions can use the long immediate source format. Conditionals, bit sets, shifts, trap, branch, and next can only use a source operand in the range 0 through 31, so they can only use short immediate. Also, the tandem form

of memory instructions can only use short immediate in the left-hand source.

Those instructions that can use long immediate format distinguish the two immediate formats based on the contents of bits 5 through 9. If the bits are all ones then it is a long immediate and the source operand is in another word of memory. But if they are any other combination (0 through 30) then it is a short immediate and the source operand is the contents of bits 5 through 9. By contrast for those instructions that do not use long immediate the value 31 has no special significance; it is just another short immediate value.

The source operand for a long immediate instruction follows the word containing the instruction pair. RISC fans, don't despair! The Gordotron still takes only one cycle per instruction. Using a wide bus, we fetch a whole block of code into a wide instruction register in one cycle. Therefore when we start to execute the instruction the instruction register already contains not only the instruction but also the long immediate value.

It is possible for both the left and right instructions to use long immediate format. When that happens, the source operand for the right-hand instruction comes first, followed by the source operand for the left-hand instruction. This is consistent with a little-endian byte order. The following diagrams depict all the variants.

Long immediate source format	Register or short immediate source format
↖ Source operand for left instruction	

Register or short immediate source format	Long immediate instruction source format
Source operand for right instruction ↗	

Long immediate source format	Long immediate source format
Source operand for right instruction ↗	
↖ Source operand for left instruction	

When any instruction reads the PC it sees the address of the next instruction. Hence if it uses the long immediate source format it sees the location following its long immediate operand(s).

Co-processor Instructions

Co-processors often need more than two operands. In addition we need a way to select the co-processor and to present the co-processor with an op code to tell it which of its own instructions to execute. Consequently a single co-processor instruction uses all 32 bits of the instruction word.

31										16	15																	0
CPU Op Code				I	Left Source				Co-processor Select				Co-processor Op Code				I	Right Source				Right Destination						

The left op code field tells the CPU that this is a co-processor instruction. The right op code field is ignored by the CPU; instead it determines the operation carried out by the co-processor. The left destination field is re-interpreted as selecting one out of 32 potential co-processors as the target for the

Note: co-processors may choose, entirely at their own discretion, to re-interpret either source field and/or the destination field. For example, for many operations the OVPU re-interprets the destination field as selecting an object buffer. However, while the co-processor can do anything it wants with its own internal registers, it can only write to the CPU register selected by the right destination field.

The co-processor instruction can use short or long immediate in either or both source fields.

For the Gordotron-64 the instruction format is asymmetrical. The complete word, containing three instructions, is shown below.

The center execution unit has three register ports and operates on two source operands, writing the result to the destination. Only one of its two source operands can be immediate. Each execution unit has 6 bits for op code plus flags, as compared to 5 for the Gordotron-32. Bit 44 is not used and must be zero.

Rethinking CPU Design

Instruction coding

Principle: *...but non-orthogonality has its uses.*

There are only nine unique op codes, but most op codes are modified by flag bits. One op code is used for all implied-operand instructions, which use the destination field as an auxiliary op code. In addition one of the op codes is interpreted differently by the right and left execution units.

The combination of op code, flag bits, identity of the execution unit, and for implied-operand instructions the auxiliary op code, yields 36 basic instructions. This is enough because the instruction count is reduced three ways:

1. Immediate short mode eliminates the need for instructions such as increment, decrement, test, and clear.
2. Mapping the program counter to a register eliminates the need for absolute and relative jump instructions.
3. Executing two instructions in sync during each cycle eliminates the need for call, return, and conditional branch instructions.

Note: this section does not show special cases. For example, test is performed by comparing to an immediate zero. See the individual instruction descriptions for special cases.

Instruction coding

The op codes and their corresponding instructions are shown in the table below, where *x* stands for 0 or 1 and *yy* stands for any combination of two bits except 00.

<i>Code</i>	<i>Group</i>	<i>Execution Unit</i>	<i>Operations</i>
00000	implied operand	see below	prefetch, purge, return, trap, branch, next
0001 <i>x</i>	conditional	right ¹	if.1, if.0
	tandem	left ¹	tandem, tandem.x
001 <i>xx</i>	calculate	either	add, subtract, compare, compare.inv
01 <i>x</i> 00	setbit	either	setbit.0, setbit.1
01 <i>x</i> <i>yy</i>	shift	either	shift.left.0, shift.left.1, shift.left.msb, shift.right.0, shift.right.msb, shift.right.lsb
10000	encode	right	encode
	co-processor	left	co-processor
10 <i>x</i> <i>yy</i>	boolean	either	and, or, xor, and.inv, copy, copy.inv
110 <i>xx</i>	optional ²	either	mul, mul.signed, div, div.signed
1111 <i>x</i>	memory	either	load, store

¹ The choice of right for if and left for tandem is constrained by a series of interlocking design decisions. First, it is natural, given Arabic-style numbers, to make the left side the more significant side in tandem arithmetic. Therefore it is natural for the 64-bit operation to set the left result flags. However, we also need the following multiply step to work:

```
add R1, R2; if.1 1, R0
shift.right.c 1, R2:R3 // shift out LSB, shift in SR.c.left
```

In order for the tandem shift to pick up the carry-out from the preceding add, the add has to be done by the left unit. That forces the if operation to use the right-hand side. Since if and tandem share an op code, that dictates that ifs are executed by the right processor and tandems by the left processor.

² Support for integer multiply and divide is optional. If supplied these, plus encode and shift, yield reasonably efficient floating-point operations even without floating-point instructions as such.

The following table shows the coding for implied-operand instructions.

<i>Bits 4 ... 0</i>	<i>Name</i>	<i>Source</i>	<i>Execution Unit</i>	<i>Description</i>
00010	prefetch	memory address	either	add block containing memory address (or next block) to the instruction cache
00011	purge	memory address	either	purge block containing memory address (or all blocks) from the instruction cache
11100	return	N/A	left	return from interrupt; copy context latch to R0 ... R31
11101	trap	5-bit trap selection	either	force a trap to occur
11110	branch	5-bit target	left	fast branch within the current code block
11111	next	5-bit target	left	compact branch to the next code block

Code map

Individual instruction op codes are shown in the table below.

	...11	...10	...01	...00
000...	if.1	if.0	reserved	<i>implied-operand instructions</i>
	tandem.x	tandem		
001...	compare	compare.inv	subtract	add
010...	shift.left.1	shift.left.0	shift.left.msb	setbit.0
011...	shift.right.msb	shift.right.0	shift.right.lsb	setbit.1
100...	and.inv	copy.inv	and	encode
				co-processor
101...	xor	or	copy	reserved
110...	div.signed	div	mul.signed	mul
111...	load	store	reserved	reserved

Note: For more information about individual instructions see the instruction reference starting on page [53](#).

Gordotron-64

The Gordotron-64 instruction set is a superset of the Gordotron-32 instruction set. This is made possible by the op code field being extended to 6 bits. The added instructions include a set of floating-point operations.

The Wide Bus

On-chip it is possible to build a very wide bus. Previously this has not been exploited in CPU design, even though suggestions to do so go back probably 15 years.⁷

People tend to think of a computer as a device for doing calculations, hence they concentrate on getting data quickly into and out of the ALU. However for most applications the bulk of processing is not calculating but replicating and rearranging data. For these applications it makes more sense to devote hardware resources to moving large chunks of data around quickly. Often there is no need for the data even to pass through the CPU. The critical metric in this case is the width of the data bus.

The Gordotron-32 has a bus width of 1024 bits, or 32 words. All data transfers between memory and other functional blocks occur over this wide bus. On chip (or on the same 3D chip stack) this is a parallel bus. Off chip it can use any packet-based bus, all of which are partly or entirely serial.

A read or write does not have to affect a full 32-word block. It is possible to read or write a contiguous subset of a block. Off-chip this may be done by varying the packet length. On-chip there is a separate enable line for each 32-bit word of the bus.

For smaller chunks of data the CPU has load and store instructions. A solo load instruction reads a single word from the address space into one of R0-31. A solo store instruction writes a single word from one of R0-31 into the address space. But, load and store instructions can also operate in tandem mode. In this mode the CPU accesses any two words within the same physical block during a single memory cycle. There is no need for the two words to be contiguous.

Thoughts about memory granularity

When the PDP-11 came out in 1970 the ability to address and manipulate bytes directly in main memory was a much-needed feature. Nowadays it is not so necessary. Now we typically use Unicode rather than ASCII, and the natural representation for Unicode is 32 bits per character.

It is still necessary to manipulate streams of bytes sometimes, but the Gordotron-32 can extract a byte – or a field of any arbitrary length – in a single cycle (one instruction pair). It can even efficiently extract fields that cross word boundaries, although this takes another half-cycle (1½ instruction pairs). Moreover, as you will see below in the discussion of the OVPU, the Gordotron can manipulate byte strings natively – just not in main memory, and not using the CPU.

Considering these factors, it is not necessary to address bytes in main memory. On the other hand, setting the granularity to one word rather than one byte expands the address space to 4 gigawords or 16 gigabytes. This is a win in itself. And, it makes it possible to segment the address space in a simple way, and still have decently sized segments.

⁷ See the Usenet group comp.arch.

Code Blocks

Principle: *It is better to parallelize than to apologize.*

The Gordotron-32 instruction register is 32 words long. It loads an entire block of instructions at a time. This speeds up processing because instruction decode delays occur only once, when the block is loaded, rather than with every cycle. Also, it means that long immediate values are already present in the instruction register, so there is no delay incurred by the use of long immediate. However code generators must be aware of and watch out for block boundaries. Hence code in main memory is grouped into physical blocks, each 32 words long and starting at an address which is a multiple of 32.

To take full advantage of the long instruction register, the instruction set contains a `branch` instruction for branching within the current code block. This instruction affects only the low 5 bits of the PC. Since the instruction cannot alter the upper 27 bits, a branch instruction cannot force the loading of a different code block, so it does not incur any delay.

When the CPU reaches the end of the code block, it loops back to the beginning of the block and continues executing there. To transfer control to another block the program must execute a jump instruction.

Note: The instruction set terminology makes a distinction between branches and jumps. A jump instruction can go to another block whereas a branch instructions can only go to a target in the same block.

Looping back applies equally to long immediate operands. For example, the last word in the block could be an instruction pair that requires two long immediate values. Those values would be taken from the first two words of the same block. Assuming no trap, interrupt, branch or jump takes place, the next instruction pair would be taken from the third word of the same block.

Instruction cache

There is an instruction cache. While the code block is in the cache, the cache intercepts and serves all requests from the CPU to load the instruction register from that code block. Note that it only affects instruction fetches, not any other kind of memory access.

The instruction cache does not need to be very big. Conventional architectures require more bandwidth for fetching instructions than main memory can support, so they have to use large amounts of cache to reduce that bandwidth. This is not necessary for the Gordotron-32. Because of its compact code and wide bus it needs only about 25% of main memory bandwidth to keep it fed with instructions, even for straight-line code. The only reason for having an instruction cache is to reduce latency.

The cache physically holds at least 64 code blocks (8K bytes), but it may be bigger. The amount of instruction cache available to any particular program is variable from 0 to 32 code blocks (up to 4K bytes) in powers of two. It is normal for each foreground process to have its own separate instruction cache. This ensures that after being preempted by a higher priority process, when it resumes processing it does not incur a cache refill delay. Hence its execution time is predictable. All background processes share a single cache of 32 objects. When scheduling a new background task, the operating system has to either reload the instruction cache or completely purge it. This is required because the cache operates on logical addresses (prior to mapping through the MMU), not physical addresses.

As a special case, instruction blocks in segment 0 are not cached. Segment 0 is reserved for physically-mapped registers and high-speed RAM so it is not appropriate to cache. Also, interrupt service code

runs in segment 0, so ignoring segment 0 instruction fetches prevents the cache for the interrupted program from becoming filled with irrelevant system code.

Since we can get away with a small instruction cache it is practical to implement in hardware a least-recently-used strategy. Code blocks in the cache are threaded on a list. When the instruction register is loaded, if the code block is not already in the cache it is added and the code block at the end of the list is purged. If the code block is already present it is merely moved to the head of the list.

A processor may support other cache strategies, but if so it must also support least-recently-used. There is a strategy register which, in the base implementation, always reads as 0 and ignores writes. Other cache strategies would be enabled by non-zero values in this register.

There is a prefetch instruction which can be used to pre-load a code block into the cache. This takes place in parallel with normal code execution. There is a queue of up to four pending prefetch operations. If the queue is full and another prefetch is executed the CPU stalls until the queue has room for the new operation. If the code block is in segment 0 the prefetch instruction silently does nothing.

The prefetch instruction ignores the low 5 bits of the address supplied to it so that you can use a jump target as-is. For example, a subroutine with the return address in R29 can use `prefetch R29` to pre-load the code block it is going to return to.

There is a special form of the prefetch instruction, for which the assembler mnemonic is `prefetch.next`. This causes the prefetch instruction to load the next sequential block after the currently executing block of instructions into the cache.

If the code block referenced by the prefetch instruction is already in the cache, the block is merely moved to the front of the list.

The instruction cache manager maintains internal state to keep track of which portion of the cache is currently being used (by the current program), what is in the cache, how recently each object has been used, and what operations are pending. Some of this data is exposed as a block of 32 words at a fixed address; this data must be saved and restored when switching between contexts. The starting address for these vectors is known as `sys.icache`. The cached content itself is not exposed.

Blocks are indexed by logical address, not physical address. Hence when switching between processes which have different segment mappings the operating system must either restore the appropriate cache context or purge the cache entirely. When an interrupt or trap occurs the CPU synchronizes with the cache by stalling until all pending cache operations are either complete or preempted. If a cache load was already in progress it completes, but other operations waiting in the queue are preempted. The queue is stored as part of the cache internal state so when the context is restored the preempted operations are re-queued.

There is a purge instruction which can be used to expel a code block from the cache. This makes space available for another code block that you want to load. (But normally you just let the least-recently-used get expelled.) Its main use is by the operating system, when freeing or altering memory containing code. Hardware does *not* detect changes to code blocks in memory and update the cache. On the contrary, the copy in the cache is static. Purging it from the cache ensures that when the code block is re-entered it will be reloaded rather than executing a stale image.

When a code block is first loaded into the cache, the MMU must have confirmed it was executable or the load would have failed. However when a code block is retrieved from the cache, its execute permission is not re-checked (the MMU is bypassed). Hence any time the operating system revokes execute permission from a block it should purge that block from any cache that might contain it.

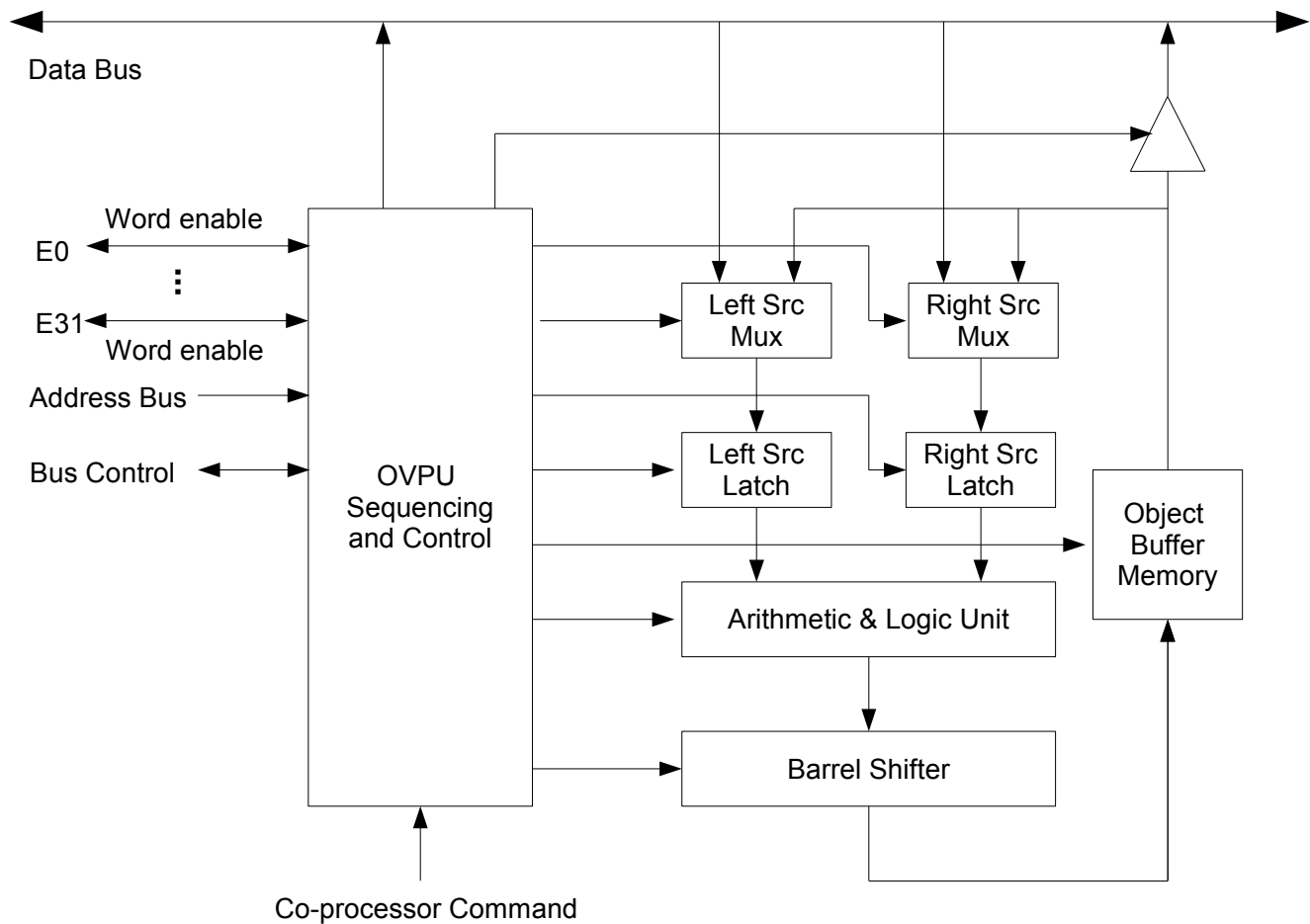
There is a special form of the purge instruction, denoted in assembler as `purge.all`. This purges the portion of the instruction cache currently in use, leaving it empty. This does not affect the instruction register, which is separate from the cache. `Purge.all` also deletes any pending operations from the queue. When the operating system creates a new cache context it first sets `sys.icache.location` and `sys.icache.size` and then executes a `purge.all` to initialize the internal state for that context.

Gordotron-64

Each code block for the -64 is 64 words of 64-bits long (512 bytes). The greater length reduces instruction-fetch bandwidth to perhaps 12% of main memory bandwidth. It also makes it more likely that your inner loop will fit into a single block.

The instruction cache holds a maximum of 64 code blocks per process. The minimum total cache size is 128 blocks (64K bytes) but it can be bigger.

Part 2: OVPU Architecture



Data Objects

Principle: *Modern software design is terrible. Modern software design is wonderful. Long live modern software design!*

The CPU described in the previous sections would be a good machine by itself, although it would need a data cache to achieve high speed. In this section I propose a more intelligent replacement for data cache, the object/vector co-processor (OVPU). A machine equipped with an OVPU can move a lot of data around in a hurry. This high bandwidth, and modest power consumption, makes it eminently suited for use in a data center. In addition the vector functionality of the OVPU can speed up a video coder or decoder, addressing the cellphone and notepad markets.

The OVPU is separate from the CPU and could be used with other CPU architectures that support co-processors. It does however require the wide bus.

Background

In the early days of Fortran, storage was done with arrays of simple data items. If you had to store three different numbers for each record, you used three separate arrays. Cobol by contrast handled composite data (records) simply because that's what was on punched cards.

Eventually the use of structures spread to Fortran and it has become integral to all modern programming languages, even procedure-based languages like C. It is of course fundamental to object-based and object-oriented languages.

In one way this is very bad. Object-based programs make a lot of memory references, and memory is the bottleneck in today's world. But it is good too. Such programs have excellent locality of data reference.

Today's typical architectures do not take advantage of that locality. They use puny cache line sizes such as 64 or 128 bits. Such a tiny cache line is unlikely to hold a complete data structure. In addition the loading and replacement of cache lines is done by dumb hardware without any knowledge of the context.

Worse, in conventional architectures there is no real support for objects. The program has to reduce many low-level operations to memory accesses at computed addresses. This should not be necessary. You should be able to load a complete object into a register with a single operation.

Some computers have provided better support for in-memory data, but only for arrays or vectors of numbers. Crays support long vectors of thousands of bits. Because an entire vector can be processed at once there is a great deal of parallelism and consequently a great increase in speed. Intel and AMD, by contrast, support what are called short vectors of only 128 bits. The latter are limited by the size of their cache lines.⁸ The ARM does not currently do vector processing at all.

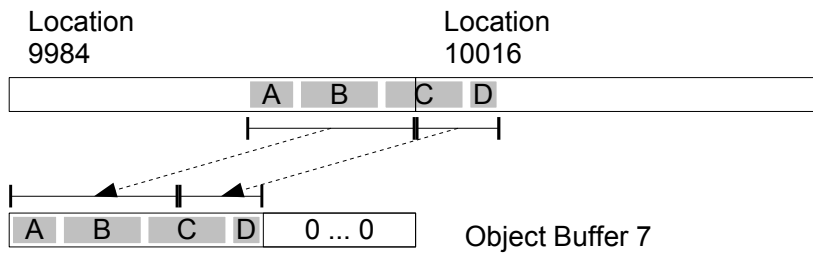
Object buffers

The Gordotron-32 has what I call object buffers. Each object buffer can hold one data object, which can be up to 32 words long.

Object buffers can be manipulated by commands to the object/vector co-processor. An object up to 32 words long can be read into or written out of an object buffer in a single main memory cycle (two if the

⁸ The recently announced next generation from AMD supports 256-bit vectors, which probably means it has 256-bit cache lines. This is a baby step in the right direction, but 256 bits is still tiny for a 64-bit machine.

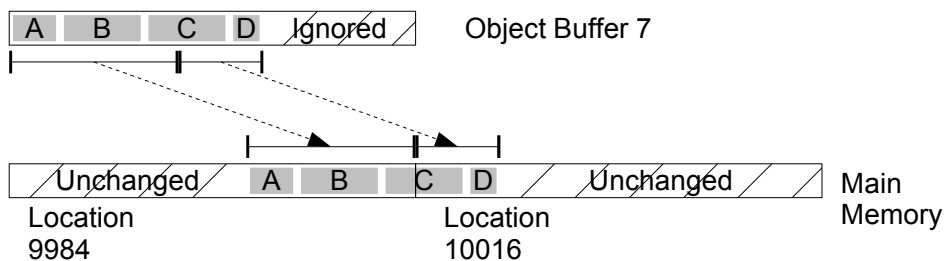
object spans blocks in main memory). Object get and put operations can proceed in parallel with most other operations, hence a savvy program can start fetching an object in advance so that when the object is needed it is already in an object buffer.



The instruction `ov.get 9999, 23, BF7` (illustrated above) passes the incoming data object through a shifter so that when it is in the object buffer it is left aligned. This is accomplished by splitting the 32-bit object address into a 27-bit block selector and a 5-bit shift. Hence if the address of the object is decimal 9999 it resides in the 312th physical block and is shifted left 15 words when loaded into the object buffer. If this object is more than 15 words long it straddles blocks and the get operation consumes two memory cycles. If the object is less than 32 words long the additional words in the buffer are set to zero.

Each object buffer is visible at a fixed place in the address space, aligned with a physical block boundary.⁹ Once a data object is in an object buffer the code can load and store properties *without having to do address calculations*. Because of left-alignment, if property D is the 23rd word in the object, it is also the 23rd word in the object buffer. But each object buffer is accessible through a fixed address, so the compiler can hard-code the address of the property. Also, using a tandem load or store, it can copy two words at a time into or out of the object in a single cycle.

The address spaces of the object registers are contiguous to each other. Being contiguous makes it easier to deal with objects that are bigger than 32 words and contain arrays. If a program loads such an object into two or more successive object buffers, it can use the normal calculation to index into the array.



The instruction `ov.put 9999, 23, BF7` (illustrated above) does the reverse shift when storing the object. The put instruction writes only as many words as specified. This length limit on writes is necessary so that objects can be kept separate. A program can put two items that reside within the same physical block into different object buffers. As long as the items don't overlap in memory the object buffers do not interfere with each other.

You can specify a different length for the put than you did for the get. This does not affect what is stored in the object buffer, only what gets written to main memory. Also, load and store instructions can access any word in the buffer. Hence you can assemble and/or modify variable-length objects in object buffers.

⁹ In segment 0, so it is never relocated by the MMU.

Architectural implications

The `ov.get` and `ov.put` commands, in combination with the wide bus, make it possible to move large chunks of data quickly. The Gordotron can push much more data around than a traditional architecture with the same clock rate and memory speed. For example, copying 1024 bytes from one location in 14 nanosecond memory to another location in 14 nanosecond memory takes 16 instructions and about 250 nanoseconds.

Object buffers take the place of L0 data cache. However unlike regular cache they are allocated and managed entirely by software.

It might seem a nuisance to have to explicitly load and store object buffers, but it is not so bad. The same algorithms currently used for allocating CPU registers at compile time can be used to allocate object buffers. And there are no additional data transfers; in conventional designs the transfers still take place, but they are hidden inside the caching hardware. This approach does take additional instructions, but not many because they are at the object level rather than the property level. At the same time it has the advantage that it uses high-speed memory much more effectively. Hardware caching at run time is terribly ineffective compared to software allocation at compile time. Hence we can either reduce the amount of high-speed memory or make better use of the same amount.

Another advantage of software management of the object buffers is that it becomes possible to predict exactly how many cycles an execution path will take. That is to say, it becomes possible once again to design reliable real-time software.

Yet another advantage of software management of object buffers is that it is not necessary to implement cache sniffing. If the compiler knows that a piece of data may change asynchronously (in C terminology it is “volatile”) it can simply opt not to use a copy of it but instead to access main memory each time it needs the data.

Object buffers are physically located in on-chip high-speed RAM. A program can have up to 32 object buffers accessible to it. The number of object buffers currently in use is determined by a 3-bit register called `sys.ovpu.bufcount`. The operating system can allocate 2, 4, 8, 16, or 32 object buffers to any given program, by setting `sys.ovpu.bufcount` to 1, 2, 3, 4, or 5 respectively. The offset of object buffer 0 is determined by a 22-bit register called `sys.ovpu.bufoffset`. The operating system can allocate private object buffers to particular programs, or it can share object buffers between programs (necessitating save and restore on context switch). The physical address for a particular object buffer is determined by bitwise ORing the buffer number, shifted left 5 bits, with the contents of `sys.ovpu.bufoffset`.

For background processes it is normal to share a single set of 32 object buffers, saving the content to and restoring it from main memory during task switches. But for each foreground process it is normal to allocate a dedicated (perhaps smaller) set of object buffers, avoiding the save and restore overhead. For really critical sections, if the program is privileged, you can place the code and data in dedicated high-speed RAM and avoid even the get/put overhead. For the actual code to enter a context see [page 51](#).

Object buffers do not require tagged RAM or anything like that, just conventional RAM. Any high-speed RAM left over after allocating object buffers is available for use as general-purpose storage.

Stalls

When transferring data into or out of main memory, get and put are asynchronous. There is a 1-deep pipeline between main memory and the object registers. When the put or get operation starts, if the

main memory is currently busy, the CPU stalls until main memory becomes idle. However once main memory is available the put or get operation starts a memory cycle and then allows the CPU to proceed. Typically it will take multiple CPU cycles before the put or get completes. During this time the CPU will stall, waiting for the put or get to complete, if and only if it encounters one of the following.

- Any instruction that references main memory.

- An instruction that reads from the same object buffer as the pending get.

- An instruction that writes to the same object buffer as the pending put.

When an interrupt or trap occurs the CPU synchronizes with the OVPU (and all co-processors) by stalling until all pending operations are complete or preempted. The OVPU may elect to preempt slow operations like multiply and divide. However if it does so it will also save the pending operation in its context data, so that when the context is later restored it will automatically restart the operation.

For any operation that writes to a destination register (R0 through R30)¹⁰ the OVPU operates synchronously. The OVPU stalls the CPU until the destination register is up to date.

Gordotron-64

The -64 uses a 4096-bit bus (64 words of 64 bits). Object buffers are also 64 words long. This is also 512 bytes so it matches up to a lot of legacy data formats.

¹⁰ Recall that the right-hand destination logic can supply the value of R31 but it cannot write to R31. This is the destination field used by the co-processor instruction, so co-processors cannot write to R31. This is intentional.

Design of the Object/Vector Processing Unit

Principle: *Don't funnel data through the CPU when it is not necessary.*

There is an object/vector processing unit (OVPU). This always appears to the CPU as co-processor 0. The OVPU performs a variety of additional operations besides loading and storing objects. As a vector processor it performs operations useful for processing strings, for encoding and decoding video, for searching for patterns, and a variety of other operations that must be repeated across multiple independent items. The main advantage the Gordotron vector commands have over i86 vector commands is a vector eight times as long. The i86 vector length is limited by its tiny cache line size of 128 bits. The Gordotron's massive bandwidth gets a lot more done in a cycle, baby.

The OVPU supports 8-bit and 16-bit data items as well as 32-bit data items. In main memory each object (including padding) must occupy a whole number of words (except for UTF-8 strings), but inside the object the co-processor can operate on smaller items.

The block diagram shown is not quite a minimal-gate-count design. One could locate all high-speed buffers in ordinary high-speed RAM. Doing so would not greatly slow down execution, because as shown the object buffer memory is still only single-port. However it would greatly increase traffic on the bus and somewhat increase power consumption.

Although the co-processor is most often used with object buffers, it can also be used with any aligned block of memory, whether high-speed memory or main memory. Whenever this documentation refers to a *block selector*, it means a 27-bit value that selects either an aligned block of memory or an object buffer. Block selector values 0 through 31 are mapped to object buffers 0 through 31. Any larger number is interpreted as the address of an aligned block of memory. On the other hand when this documentation refers to a *buffer selector* it means a 5-bit value that selects one of the object buffers. For example, the `ov.and` instruction performs the bitwise AND of two vectors and places the result into a third vector. The syntax is defined as `ov.and block, block, buffer`. This means that either source vector can be in an object buffer or in any aligned block in memory. However the destination must be written to an object buffer. (You can of course specify the same block buffer for the destination and one or both sources.)

The OVPU contains a 128-bit selection set register. This register is used to condition certain commands so that they only alter a subset of the items in the target object buffer. When an item is included in the selection set the corresponding bit in the selection set is 1; when the item is excluded the corresponding bit is 0. However for items bigger than 8 bits each there is more than one selection set bit per item. In this case the item is included if any bit that applies to it is set.

The OVPU can accomplish string processing. The general approach is the same as processing bit fields within the CPU; you have to shift the strings into alignment, exclude the characters you don't want and merge the rest together. However, there is no necessity for an explicit mask operation. Instead, you can apply a selection set to an `ov.copy` command so that it only copies the substring you want. Also, you can combine shifting and copying into one command (in fact `ov.copy` is actually a shift with a shift count of zero). Hence if your selection set is already set up you can append or replace a substring with a single operation.

There is a 32-bit status register which is only writable with privilege. The content of this register is as follows:

31	30	29
Error	Status	

The 2-bit status field reports the overall status of the OVPU.

<i>Value</i>	<i>Meaning</i>
0	The most recent operation completed successfully.
1	Bad data. Currently this can only be set by a UTF-8 conversion error.
2	Unrecoverable error (e.g. bus error). The pending operation was abandoned.
3	Internal error. Hardware failure.

The error bit indicates that a serious error has occurred. The status field contains 2 or 3. The error can be reset by a master reset or by writing all zeros to the status register with privilege. Either way the OVPU may take some time resetting itself.

There is a 32-bit mode register which is both readable and writable without privilege. The content of this register is as follows:

6	5	4	3	2	1	0
UTIE	Item Size		Overflow		Subset	

The 2-bit subset field specifies which subset of words are to be operated on. The selection set is held in a 128-bit selection set register which is internal to the OVPU. The subset is specified as: 00 – all items, 10 – selected items only, 11 – non-selected items only. The combination 01 is not used in the subset field.

The 2-bit overflow field controls the handling of overflows. When an item overflows its allocated size (for example because an add operation generates a number that won't fit in the item) any of three things can happen. 00 specifies that nothing is to be done; the excess bits are just discarded. 10 specifies that the result is to be limited as an unsigned number; overflow goes to all ones and underflow to all zeros. 11 specifies that the result is to be limited as a signed number; overflow in the positive direction goes to the most positive number (01 ... 1) and overflow in the negative direction goes to the most negative number (10 ... 0). The combination 01 is not used in the overflow field.

The 2-bit item size field specifies the size of the items when the block of data is treated as a vector. Item sizes are coded as 00 – 32 bits, 01 – 16 bits, 10 – 8 bits. The combination 11 is not used in the item size field.

The UTIE field, when set to 1, enables interrupts to be generated by UTF-8 conversion errors. See the instruction descriptions for more detail.

The above-mentioned registers are also exposed as high-speed device registers (sys.ovpu.status, sys.ovpu.mode, and sys.ovpu.select1...4) in segment 0, where they are only accessible to privileged code. This is done to allow fast save and restore of the co-processor state during a context switch. These registers are in addition to sys.ovpu.buffcount and sys.ovpu.bufoffset, described in the previous chapter.

Some OVPU instruction may be rather slow. They may also trigger page faults that need to be handled by software. Either way it is necessary for the current OVPU operation to be preempted and then to restart when the context is restored. There are additional words of internal status that are required to

restart after an operation is preempted; when all these registers are restored the OVPU automatically restarts the preempted operation. This additional internal state includes the command register (5 bits), source and destination fields and source immediate flags from the co-processor command (17 bits), and contents of the CPU operand latches during the co-processor command (3 words), plus miscellaneous internal state (> 1 bit).

Commands for the OVPU come in six groups: register commands, memory-access commands, vector-to-vector commands, shift commands, single-vector commands, and selection set commands.

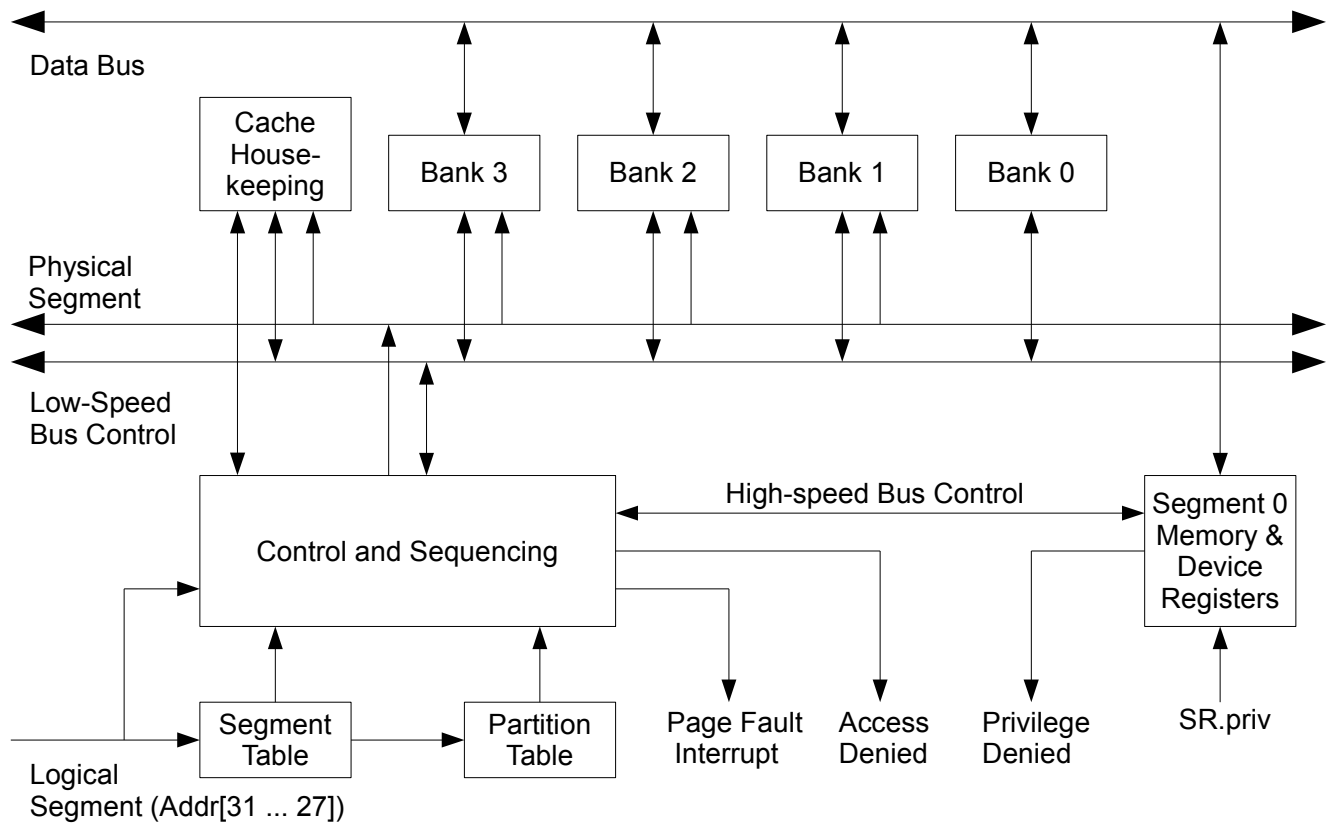
OVPU commands are summarized in the following table.

<i>Group</i>	<i>Op Codes</i>	<i>Commands</i>
register	0, 1	read, write
memory access	2 – 7	get, put, getutf8, pututf8, getitem, setitem
vector-to-vector	8 – 14	add, sub, mul, div, and, or, xor
shift	15 – 18	shift.left, shift.right, rotate.left, rotate.right ¹
single-vector	19, 20	fill, resize
selection set	21 – 26	value, compare, subset, saveset, include, exclude

¹ copy is a synonym for shift with a count of zero.

Different groups interpret the fields and operands of the co-processor instruction differently. In addition different groups pay attention to different fields within the mode register. For detailed information about each command, see the OVPU command reference starting on page [76](#).

Part 3: Memory Management



Memory Management

Just lately there have been important advancements made in the use of verified code. Java was the first large-scale implementation to rely upon just-in-time compilation of verified intermediate code. This technology has been improved and now Google claims they can do it even for non-object-based languages like C and Fortran with a modest overhead – under 5%, according to some reports. If this proves to be the case then we do not need memory protection anymore. Software enforcement of object boundaries not only has a lower overhead, it also does a much better job. Hardware memory protection only works at the block level. It does not catch a program going beyond the end of an array if it only tramples some more of its own memory.

However, while memory protection may not be needed any more, we still need memory relocation. In particular, we need to support caching. You could potentially abolish the remapping of addresses that normally is done for virtual memory, but you cannot realistically abolish the hardware support for searching and quickly returning data which is already in main memory. There is no way to do that fast enough in software. And it turns out that, if you are going to do that much in hardware, you might as well do remapping and protection.

These days most CPUs rely upon page-based memory management. In such an architecture every view of memory is flat. This is simple conceptually but it causes high operating-system overhead. Every time a process wants to map pages into its memory space, for example so it can use a code library, the kernel has to provision a whole bunch of page mappings into the flat process space. And whenever a context switch occurs there are many page mappings to be flushed and reloaded. This is why Windows and Linux programs often have quick-start arrangements whereby they are provisioned at boot time. This is the same hack that was employed in ye olden days on the VAX, and for the same reason.

The Gordotron-32 instead uses a segmented approach. Stop your groaning! Segmentation acquired a bad name because it was used as a hack to extend the address space of 16-bit processors. But when it is used correctly (as it was on the PA-RISC) it is a simple, clean way to manage memory.

The address space is divided up into 32 segments. Each segment holds at most 128 megawords or 512 megabytes. This is a pretty decent size in itself, but if a single larger space is needed, consecutive segments can be set up to access contiguous memory.

Ordinary unprivileged processes can only access the segments set up for the process by the kernel. Each segment points to a physical or virtual region of main (not high-speed) memory. Each segment has individual permission bits for read, write, and execute. It points to a memory partition (described below). It also has an offset and a size, so it can give access to just a portion of the partition.

Segment Property	Size (bits)	Type	Description
read.p	1	control	read permission with privilege
write.p	1	control	write permission with privilege
execute.p	1	control	execute permission with privilege
read.u	1	control	read permission without privilege
write.u	1	control	write permission without privilege
execute.u	1	control	execute permission without privilege
partition	6	control	index of the partition descriptor
size	22	control	size of the segment (blocks)
offset	32	control	offset into the partition (blocks)

Multiple processes may share segments. This is the usual and expected implementation of shared code libraries and shared read-write memory.

The complete set of segment descriptors is exposed as three 32-word blocks of high-speed registers, collectively known as sys.seg.

Logical segment 0 is special. Segment 0 is never relocated. Instead, logical segment 0 always points to physical segment 0, which contains on-board physical devices and high-speed memory.

Other logical segments cannot be mapped to physical segment 0. There are two separate physical sets of bus-control lines for physical segment 0 and for all other physical segments, although the data lines are shared. Anything that is not part of physical segment 0 is accessible only through the MMU. At boot-up the kernel must set up the segment table before it can access anything outside physical segment 0.

When the privilege bit (SR.priv) is set all of segment 0 is accessible, regardless of the segment 0 descriptor. Without privilege most of segment 0 is inaccessible except through the segment descriptor, however there may be particular blocks that are accessible regardless – each block in segment 0 manages its own access rights. By contrast access to other segments is governed only by the segment table, so it is always the same within a given segment.

Instead of size and offset properties, the table entry for segment 0 has 27-bit low limit and high limit entries. Thus an unprivileged program can be given access to a single contiguous region within page 0, which would otherwise be inaccessible to it. This makes it possible for an unprivileged program to access a device so it can act as a driver, or for an unprivileged program to be allocated a single contiguous block of high-speed RAM.

MPU vs MCU

Traditionally there are two different approaches to handling memory, the MCU approach and the MPU approach. The MCU approach is to use internal¹¹ memory as simple RAM (usually supplemented with Flash). The MPU approach is to use internal memory purely as cache. Neither of these approaches is as flexible as one would wish.

Meanwhile, people are starting to replace ludicrously slow mechanical memory – worse latency than

¹¹ Normally "internal" means on-chip, but if 3-D stacking works as well as its proponents hope, change “on chip” to “on stack”.

the fixed head drums that were used 50 years ago! – with semiconductor memory. But they are driving that semiconductor memory with software written for driving disks, instead of with hardware that can keep up with the bandwidth of the memory. Consequently they are not achieving anything like the full potential of the hardware. (Currently the controllers for “semiconductor disk” are quite slow too, but that will change as expectations rise.)

With semiconductor memory being used as backing store, the distinction between real and virtual memory is fading. The main distinction we need to make is whether or not a partition is cached.

This design unifies the MPU and MCU approach. Internal low-speed RAM (not segment 0) can be used as a mixture of ordinary RAM and cache, in configurable proportions.

When caching, data by default is paged in and out in 1024-bit blocks. This block size ought to be acceptable for semiconductor memory, however any partition can cluster blocks together. In this mode the MMU allocates a contiguous set of cache rows to represent one cluster. For example, if the cluster size is 4 the partition operates in clusters of 4096 bits (512 bytes) and the MMU uses four successive cache rows for each cluster.

When backing store is a disk, or otherwise hard to talk to, cache misses are delegated to software. Instead of stalling on a page fault the current instruction is aborted and a page-fault interrupt is generated. This interrupt can be used to reschedule. For loads and stores there is no need to back out of the aborted instruction because the CPU never updates its state until the last clock phase (the X2 phase), and page faults are always detected earlier than this. When a page fault occurs while loading the instruction register (phase X0), the instruction register ends up containing all zeros. This doesn't matter, however, because the pending page fault interrupt occurs at the start of X1, as all interrupts do.

Failed co-processor operations are a different matter. These may proceed in parallel with the CPU so by the time they fail the CPU may have already moved on to another instruction. Co-processors which operate in parallel with the CPU (such as the OVPU) expose enough internal data so that they can restart any particular operation after a failure. This data is part of the internal co-processor status information that is saved and restored by the operating system during context switches. Hence when the context is restored the co-processor retries the operation.

There is a special case that occurs when the CPU is stalled pending the completion of a co-processor operation and the co-processor operation cannot complete because of a software-mediated page fault. In this case the co-processor releases the CPU (stops stalling it) but not until the page-fault interrupt request has reached the CPU. The CPU was stalled in an extended X1 phase; now X2 is inhibited so the CPU does not update its state. Therefore when the interrupted context is re-entered the co-processor will restart the failed operation, while the CPU re-executes the last instruction pair and once again is stalled by the co-processor.

Partitions

Memory partitions can exist on-chip (internal partitions) or off-chip (external partitions). There may be multiple partitions on the same physical device, but partitions must not overlap.

Partitions are described in a table. For fast access this table is internal to the MMU itself, but it is accessible to the kernel as a multi-block region in segment 0 called sys.part. The table has room for 63 partitions (partition index 0 is not used). The partition table controls whether the partition is cached and details of caching. It also contains permission bits. The effective permissions for an access through a segment register are computed as the AND of the segment permissions and the partition permissions.

The partition table is expected to be relatively static, however removable volumes may come and go. The kernel must be careful when dismounting or remounting a cached partition to be sure there are no stale blocks in the cache. See the discussion below.

<i>Partition Property</i>	<i>Size</i>	<i>Type</i>	<i>Description</i>
read.p	1	control	read permission with privilege
write.p	1	control	write permission with privilege
execute.p	1	control	execute permission with privilege
read.u	1	control	read permission without privilege
write.u	1	control	write permission without privilege
execute.u	1	control	execute permission without privilege
volume	16	control	volume or device identifier (0 = internal)
offset	32	control	offset of partition from start of volume (blocks)
size	32	control	size of partition (blocks)
cache.mode	2	control	0 (none), 1 (software), 2 (write-through), 3 (full)
cache.cluster	3	control	number of blocks in a cluster (power of 2)
cache.size	5	control	number of rows in cache as a power of two
cache.address	32	control	location of start of cache (least significant 2 bits are hard-wired to zero)
cache.hits	64	status	total number of cache hits
cache.misses	64	status	total number of cache misses
cache.blocks	32	status	number of blocks currently in cache
cache.dirty	32	status	number of dirty blocks currently in cache

You can deduce from the above table that the largest allowable partition is 4 billion times 128 bytes (equals one block), or 128 Gigabytes. Of course a single process could only access a small part of this at any one time.

Example

A service is implemented as a parent process which is also the dispatcher, and multiple service processes. The parent, or dispatch, process has the following segments.

<i>Segment</i>	<i>Characteristics</i>	<i>Description</i>
1	Read/execute, fixed size	Code segment
2	Read/write, growable	Stack segment
3	Read/write, growable	Heap segment
4	Read/write, growable	Mailboxes
5	Read/write, growable	Global objects

Segments 1, 2, and 3 are private and typical for all processes. Segment 4 contains an array of mailboxes, one per service process. Segment 5 contains objects to which the service processes must have access.

Each child, or service, process has the following segments.

<i>Segment</i>	<i>Characteristics</i>	<i>Description</i>
1	Read/execute, fixed size	Code segment
2	Read/write, growable	Stack segment
3	Read/write, growable	Heap segment
4	Read/write, fixed size	Mailbox
5	Read-only, growable	Global objects

Note: In this example the segment number assignments correspond between parent and child processes. This reflects good programming practice, however it is not by any means required. Any segment number (except 0) can point to any segment descriptor, so you can mix them up any way you want.

All service processes share common code, so they share the same physical descriptor for segment 1. By contrast they each require private stack and heap so segments 2 and 3 each have unique descriptors.

Segment 4 is a special case. Each service process has to be able to read and write its own mailbox but it has no business accessing the mailboxes of other service processes. Each segment 4 is a small section of the dispatcher's segment 4. For example, assume that there are 500 service processes and each mailbox is 8 K words. The dispatcher's segment 4 is a total of $500 * 8 \text{ K} = 4 \text{ M}$ words long. Each service process' segment 4 is just 8 K words long and offset from the start of the dispatcher's segment 4 by a multiple of 8 K.

Finally, segment 5 contains global objects which are maintained by the dispatcher and readable to the service processes. The service processes share the same descriptor, which is similar to the dispatcher's segment 5 descriptor except that it is restricted to read-only access. The segment can grow but not at the hands of the service processes; it grows when the dispatcher makes it grow.

In many application designs the global objects may be updated by any thread. These applications use semaphores or other arbitration methods to enforce single-threaded updates in critical sections of code. However, this is invisible to the memory management unit. Such an application would configure the shared object segment as read/write shared memory. In the example, it would use the same readable, writable, and growable descriptor for segment 5 for the dispatcher and for each service process.

Caching

Only external partitions can be cached. Cached partitions can employ a full-caching strategy or a write-through strategy. In addition a partition can be set to generate a cache-miss interrupt when a cache miss occurs, rather than updating the cache in hardware.

Internal RAM can be used as 4-way set associative cache. Each external partition descriptor specifies what, if any, region of internal RAM it uses for caching. Multiple partitions can share the same cache or they can have dedicated cache. Each caching region must have a size, in rows, equal to a power of two.

Dedicating cache to a single partition is less efficient than sharing it, but it can be desirable because flushing cache can be a slow process. Before a partition can be dismounted it is necessary to traverse its

whole cache looking for blocks belonging to that partition and either writing them out or nullifying them.¹² Besides, at shutdown or dismount time any non-volatile memory has to be updated by flushing its cache.

Since a block contains 1024 bits of data, a read of the 4-way cache yields 4096 bits of user data. However it also has to yield 164 bits of cache housekeeping data. Hence on-chip main memory is actually 4260 bits wide. When used as a cache block all of the bits are used. However any given operation only reads or writes 1024 bits of data, so the main data bus stays at that width. There is an auxiliary bus for the housekeeping data. When used as ordinary memory the housekeeping bus is not used, and the 4096 bits of user data appear as four separately addressed 1024-bit blocks.

For each cache row the housekeeping data includes a bit pattern indicating the relative recentness of use of the blocks in order to support an LRU replacement policy within the set. (There are only 24 combinations but it is easiest to store this data as a sequence of 4 two-bit numbers). The housekeeping data also contains, for each of the 4 blocks in the set, a dirty bit, the index of the descriptor for the partition that the block is currently caching, and the logical offset of the block within that partition.

There may also be additional bus width and memory width for error detection and/or correction (EDC). This is transparent to software, except that at power-up the kernel may need to make a pass initializing RAM before enabling the EDC function.

Cache blocks which are not currently in use (free blocks) have the partition descriptor index set to zero. (This is why the partition table holds only 63 entries, not 64.) Freeing the block also marks it the least-recently-used in its set.

When the CPU requests a read of a block from a cached external partition, the MMU carries out the following steps.

1. It verifies the access permissions and limits. If it detects a violation it raises an access error interrupt and stops processing.
2. It computes the physical address of the row corresponding to the block and performs a read cycle to look for the requested block in the cache.
3. If the block is in cache, the MMU passes the block to the CPU. If the requested block is not already the most recent of the set, it performs a write cycle to update the relative recentness data. It also updates the hit count for the partition to which the block belongs. Then it ceases processing.
4. The block was not in cache. The MMU updates the miss count for the partition to which the block belongs. If the external partition is set for software caching the MMU generates a page fault interrupt and ceases processing.
5. The external partition is set for hardware caching. The MMU attempts to load the block from the external partition. It waits as long as the transfer takes. If the block is not there the MMU raises an access error interrupt and ceases processing.
6. The block was found. The MMU passes it on to the CPU, which resumes processing in parallel with the MMU. The CPU may start another request while the remaining steps are still being performed by the MMU pipeline, however it is not guaranteed that the new request will proceed without stalling.
7. The MMU creates space for the new block in the cache row by freeing the space currently

¹² To help make flushing more efficient the MMU maintains a per-partition count of total cached blocks and cached dirty blocks.

occupied by the least-recently-used block in the set. If that block is dirty in the cache the MMU writes it to external memory. The MMU updates the block count and dirty-block count for the partition to which the old block belonged.

8. The MMU copies the new block into the row, updates the recentness data, and writes the updated row to the cache. It updates the block count for the partition to which the new block belongs.

Error handling

Memory access errors can occur for various reasons. The requested address could be out of range, the particular operation could be forbidden, there could be a hardware error (bus fault), or the data could simply not be in cache and the partition uses software cache loading. In any case we want to make error recovery as simple as possible. This is particularly important for cache miss handling.

Access errors can occur only in certain circumstances. These are listed below.

If an error occurs during a fetch started by a prefetch instruction, the instruction cache ignores the error and continues with that object absent from the cache. When and if a jump targets that code block the instruction register (and cache) will attempt the load again.

If an error occurs during instruction fetch, the instruction aborts with the CPU in a consistent state and the PC pointing to the same location. Returning to the context causes the CPU to try again to load the instruction register.

If an error occurs during a load or store instruction, the instruction aborts with the CPU in a consistent state and the PC pointing to the same location. No results are written by either instruction in the pair. Returning to the context causes the CPU to re-execute both instructions in the pair.

If an error occurs during a get operation, the CPU has already gone ahead by the time the error occurs. The object manager sets the object buffer status to getFailed. Any subsequent load, store, or put instruction that uses the invalid object buffer traps. However if another get is executed and it is successful it resets the state of the object buffer.

If an error occurs during a put operation, the CPU has already gone ahead by the time the error occurs. Unfortunately, this error leaves memory in an incorrect state. The object manager sets the object buffer status to putFailed and raises an interrupt. Any subsequent load, store, or put instruction that uses the invalid object buffer traps. The object buffer can only be reset by a successful get operation.

A co-processor may encounter memory access errors. The co-processor should behave as much as possible like load and store for synchronous operations, and like put and get for asynchronous operations.

Layers of caching

People accustomed to layer upon layer of hardware caching may feel that this design does not have enough layers. The theory has been that data has to move closer to the CPU as it is used more frequently. In this design the layers are not so obvious because some of them are handled by software, and the larger cache line size means that when you do move data you accomplish more.

It may help to list all of the layers of caching.

1. The general registers.
2. The instruction cache.
3. Object buffers.
4. Segment 0 RAM.
5. Ordinary on-chip RAM used as RAM.
6. Uncached off-chip RAM.
7. Off-chip RAM cached on-chip via hardware paging.
8. Off-chip mass storage cached on-chip via software paging.

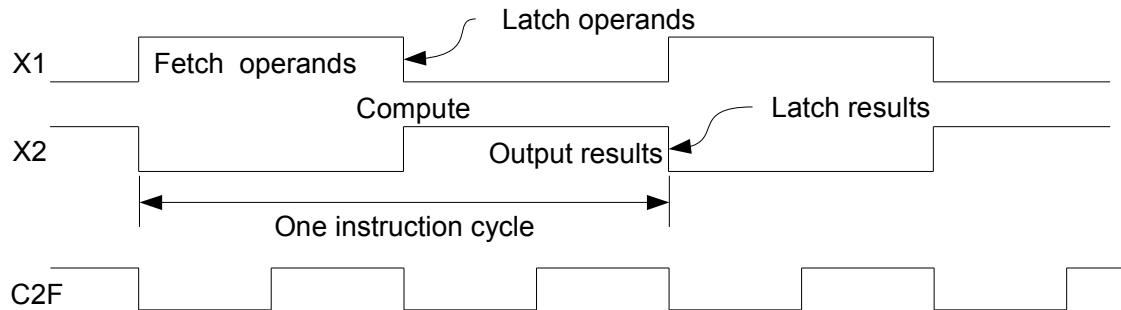
It that's not enough layers for you, don't ask me to bake your birthday cake!

Gordotron-64

For the -64 the word size is 64 bits and the total address space is ridiculously large. For a given process there are 64 segments in the segment table. Each segment can be ridiculously large.

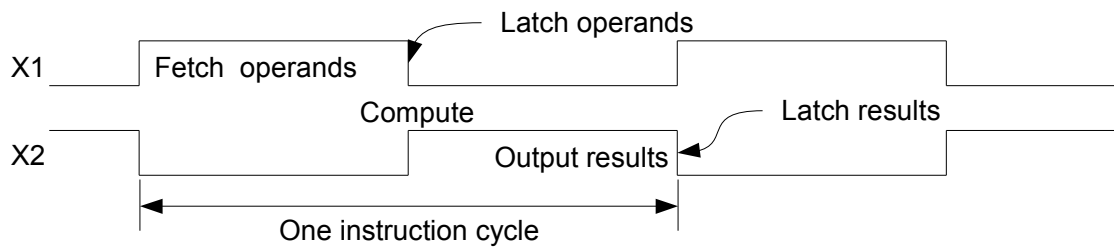
The natural cache line size for the -64 is 4096 bits, which interestingly enough happens to be 512 bytes. However larger cluster sizes are supported, as for the -32. Internally the main on-chip memory has a line size of 16384 bits plus about 300 bits, for about 2% overhead.

Part 4: The Instruction Cycle



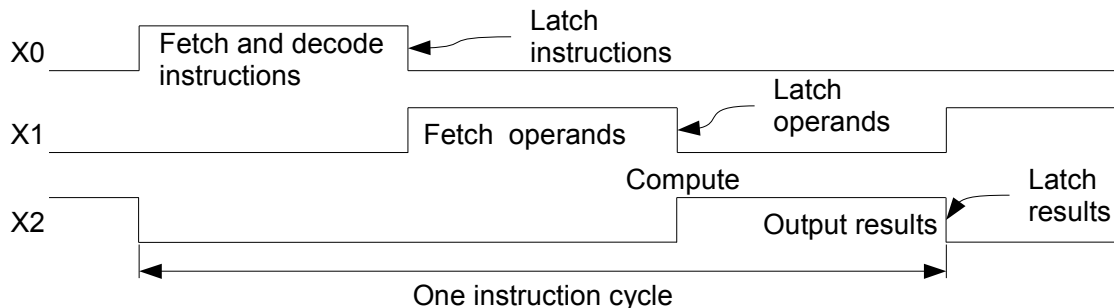
The Instruction Cycle

There are three possible clock phases, known as X0, X1, and X2. However a typical instruction only has X1 and X2, as shown below.



Note that the “compute” step vaguely overlaps both X1 and X2. This is pretty much standard. Each operand latch is a single-stage latch, not a master-slave flip-flop, so during X1 they start passing data through as soon as their inputs settle. If the ALU is ready (if it is in the appropriate mode, as driven by the X1 instruction decode) it can start processing the data right away. During X2 the register or memory address which is being driven is also a simple latch. It doesn't matter if the data being supplied to it at the beginning of X2 is incorrect, as long as it settles early enough before the end of X2 to meet the setup-time requirements for the register or memory address.

X0 is used only when it is necessary to fetch a block of instructions. This happens after a jump, an interrupt, a trap, or a master reset. In this case the instruction cycle is as shown below.



Because a normal instruction cycle consists of just two clock phases, this time duration is referred to as one clock cycle. For example if the processor runs on a 1 GHz clock then a normal instruction cycle takes one nanosecond, and the minimum duration for either X1 or X2 is 0.5 nanoseconds. However clock phases can be stretched as necessary, in units of one half-cycle (0.5 nanosecond).

We could run our oscillator faster and give ourselves a finer resolution than 0.5 nanosecond, and this would allow us to tune performance better, but high-speed oscillators burn a lot of power. In future we might be able to employ asynchronous logic, in which there is no master clock and each instruction takes only as long as it needs.¹³ Or we could employ higher-speed on-chip oscillators using MEMS technology, which might require less power.

The actual instruction fetch occupies at least one-half clock cycle, even if the code block is in the instruction cache. It may take considerably longer, in which case the responsible subsystem stretches the X0 cycle. In addition X0 stretches another half-cycle to allow time for the X0-level decode to take place.

X0 decode is a shallow decode which only parses the instructions, it does not distinguish between

¹³ There is nothing new under the Sun. Asynchronous logic was employed in computers as far back as 1951. However chip-design software that supports clockless logic has only recently appeared.

individual op codes. The function of X0 decode is to decide where each instruction in the block begins and ends, and from where its source and destination are to be obtained. This is done with a separate logic circuit dedicated to each word of the instruction register. Hence at the X0 level we perform an up-to-32-way prefetch and decode. However because the instruction format is regular and simple, and because we only need to partially decode, this takes only about 500 gates for the entire instruction register. This is far less than is normally required for a pipelined multi-way prefetch and decode unit.

X1 decode is a deep decode that examines the whole instruction to decide how to configure the ALU and other hardware. However X1 decode is only performed on the currently executing instruction, so this logic is not replicated. While X1 decode is taking place the operands are being fetched and latched using the signals provided by X0 decode.

Up to the end of X1 nothing has been written except to the operand latches, so it is still possible to back out of the instruction. For example, during a load or store operation the MMU may find that the memory location is valid, but it is in virtual memory and is not currently cached, and moreover the partition requires software page-fault handling. The MMU stretches X1 so that it can make this determination before X1 is over. Then, if it finds this particular circumstance applies, it raises an abort request before allowing X1 to end. In response to the abort request X2 is suppressed. A half-cycle time slot is still used but the X2 signal remains false. As a result the PC does not update, the destination register is not updated, memory is not altered, and result flags in the SR are unchanged. At the same time the MMU asserts the page-fault interrupt, so at the end of what would have been X2 the interrupt is serviced. Whenever the program's context is restored, the same instruction re-executes from the beginning.

Older CISC machines were unable to simply re-execute a faulted instruction because they accessed multiple memory locations with a single instruction, hence a single instruction could incur page faults at multiple phases of execution. This machine, like RISC machines generally, keeps each instruction simple enough that this cannot happen. This is why load and store instructions cannot be combined, that is, you are not allowed to put two different instructions from that group in the same word. It is true that you can access two locations at once using a tandem load or store, but the two locations are constrained to be in the same block so only one page fault can occur.

Another operation that needs to be finished before X2 starts is determining whether the condition set by a condition instruction is met or not met. By sticking with a very simple operation – just selecting a bit from the destination operand – we can do this using dedicated logic in parallel with the main ALU. But this logic is simpler than an add or subtract, and has no carry between bits, so it only takes three or so gate delays. This means that at the end of X1, while the ALU is just starting to process the data, the conditional logic has already finished its work. If the condition is not met the logic asserts a nullify request. Like abort request, this must be asserted before the beginning of X2. This is not quite the same as an abort, because the PC updates as normal, but otherwise it is like an abort. It prevents the destination or memory from being written to, and it prevents status flag updates in the SR.

Another instruction set feature that improves and simplifies the memory interface is the simplicity of the load and store commands. These do not do any calculation, and as a result the address is available as soon as the output of the source latch settles. The data path for the address bypasses the ALU. This gives extra time for the memory to decode the address, and as a result high-speed on-chip memory can achieve one-cycle performance (stretching X1 by just one clock cycle), even if the memory takes 1.6 cycles to perform a read or write operation.

Synchronizing with co-processors

Some co-processor operations may be rather slow. At the start of X0, when an interrupt request is asserted, the CPU emits a sync request to all co-processors plus the MMU. It then waits for X0 to end. However X0 may be extended beyond its normal half-cycle by any co-processor or by the MMU.

Each co-processor decides for itself how to react. If it is performing a fast operation it extends X0 until it has finished the operation, then allows X0 to finish. For a slow operation, it may suspend the operation so it can resume it later, or abandon the operation so it can restart it later.

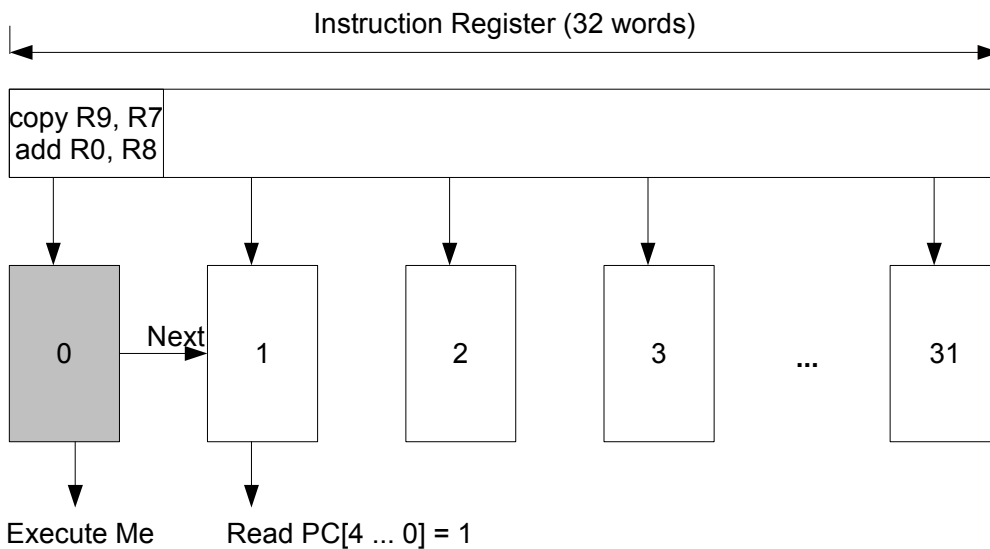
As each co-processor reaches the point at which it is ready to sync it releases the X0 stall line. When all co-processors plus the MMU have released the X0 stall line it goes to one and the X0 phase finishes.

The CPU also performs a sync operation at certain other times. It does so during a return from interrupt instruction. It may do so at other times, as determined by implementation details.

The MMU or a co-processor may assert abort request before X0 ends. If abort request is asserted when X0 ends, the CPU backs out of the instruction so it can be re-executed later. A co-processor may only assert abort request if the instruction that is currently executing is a co-processor instruction directed to that particular co-processor. The MMU may only assert abort request if a load or store instruction is currently executing, or the current memory cycle is an instruction fetch.

X0 Instruction Decode

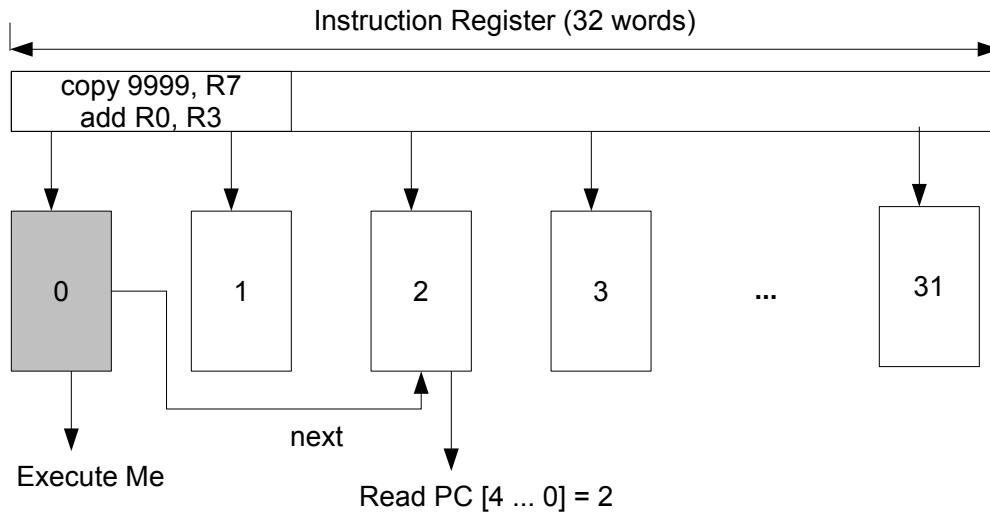
Because of the way instructions are fetched and decoded as a block, the PC is not like a normal register. The upper 27 bits are normal, but the lower 5 bits are implemented as a ring counter composed of 32 master-slave flip-flops. Each stage corresponds to one word in the 32-word instruction register. When the flip-flop is set (gray in the image below) the instruction pair from the corresponding word executes.



In the illustration, stage 0 knows that it is the currently executing instruction. It also knows, by looking at its instruction pair in the instruction register, that its instruction pair occupies just one word. Accordingly it tells stage 1 that stage 1 is the next instruction that will be executed.

The low five bits of the PC have to be readable as a binary number. This number is emitted by the stage which knows it is next (in the example above, stage 1). It is faster to do what amounts to a lookup than it would be to encode the ring-counter output into a binary number. It is important that this operation be fast because it is on the critical path.

Contrast what happens if the instruction in slot 0 uses a long immediate operand. In this case the decoder in slot 0 recognizes that its instruction takes two words. It sends the “next” signal not to slot 1 but to slot 2, as shown below.



Note that the X0 decode logic in slot 1 is trying to decode a bogus instruction (it is not an instruction but the data value 9999). This does not cause any harm because slot 1 never becomes the active slot. At the end of the current instruction cycle, control transfers to slot 2. Of course, the program can explicitly branch to slot 1, but that is a coding error which can happen on any machine with variable-length instructions.

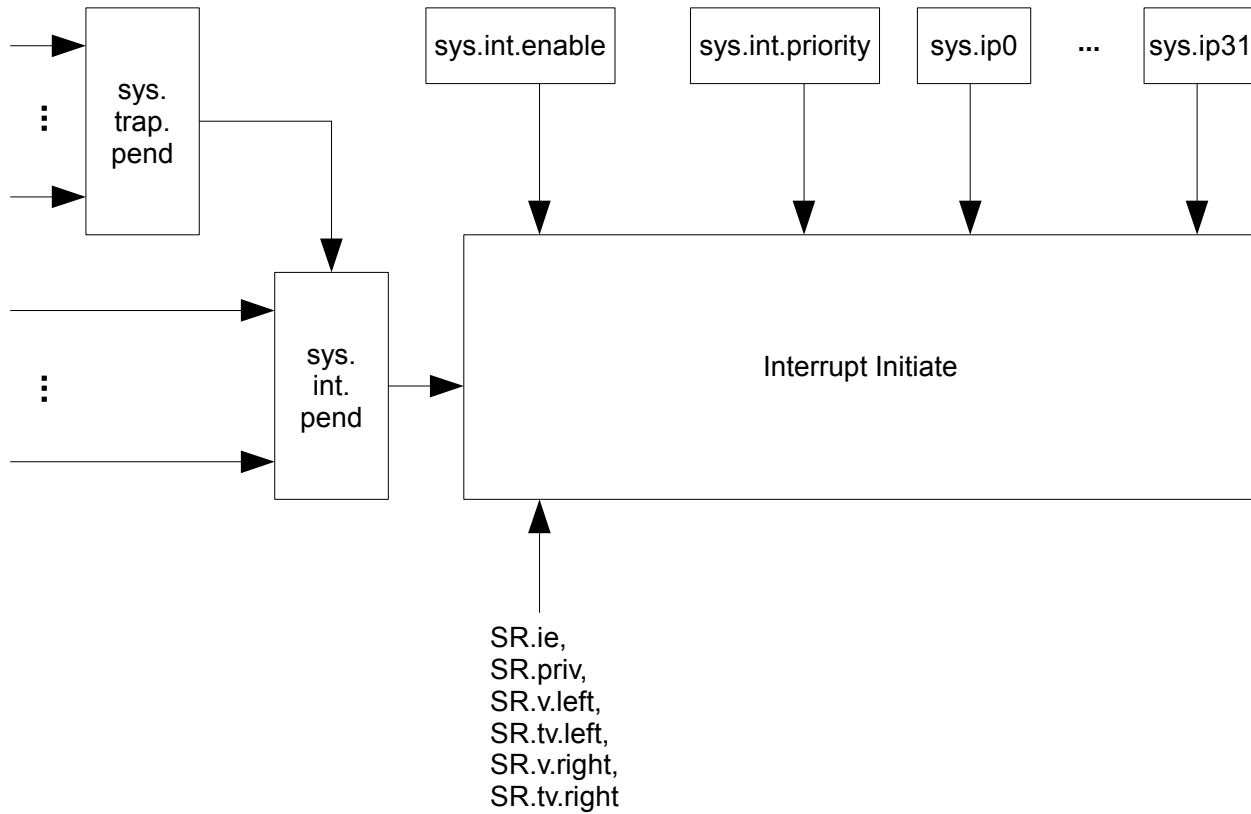
The instruction in slot 0 could have specified two long immediate values, one as the left source and one as the right source. In this case the slot 0 logic would have notified slot 3, rather than slot 2, that it was the next slot.

What happens if, for example, the instruction in slot 31 uses a long immediate? The long immediate value is taken from slot 0 of the *same* instruction block. In other words, the block wraps. The PC is consistent with this, because the upper 27 bits do not update automatically; when the ring counter wraps from 31 to 0 the PC goes back to the start of the block. Hence, execution by default loops back to the beginning of the block. To transfer control to any other block the program has to explicitly jump.

This may seem awkward, but it eliminates what would otherwise be hardware dedicated solely to incrementing the PC. I hate dedicated hardware. You should use the resources to make something more general, and then let software apply the generalized hardware to the particular task when it needs to. The rest of the time software can use the generalized hardware for other purposes. That is what happens here. The left ALU is used to perform jumps. Each block must exit with a jump, so at least 1/32nd of the time the left ALU is used for jumping. The rest of the time it is available for other purposes.

There is a compact jump instruction (called `next`) that can be used to jump to a target anywhere in the next sequential block. This is particularly useful when you want to jump into the next block from slot 31 and slot 0 is already used for other purposes. The PC has already wrapped back to the beginning of the current block, so an ordinary relative jump would need to add at least 32 to the PC. This would require a long immediate, which won't fit. Instead you can use a `next` instruction and get by with a short immediate.

Part 5: Context Handling



Interrupt Handling

Principle: *Support the programmer, don't tell him how to do his job.*

The following table shows bits in the status register that are relevant to interrupt processing.

<i>Bit</i>	<i>Name</i>	<i>Access</i>	<i>Description</i>
31	SR.priv	privileged write	running in privileged state
30	SR.ie		enable interrupts
8	SR.tv.left	read-write	enable trap on overflow flag (left processing unit)
0	SR.tv.rignt	read-write	enable trap on overflow flag (right processing unit)

When the SR.ie bit is 0 all interrupts and traps are locked out, regardless of the settings of any other flags. (A master reset still works, though.) Interrupts and traps are re-enabled by writing a 1 bit to the SR.ie bit. However this bit is only writable in privileged mode.

The CPU is in privileged mode whenever the SR.priv bit is 1. Writing 0 to it when it is 1 causes the CPU to enter unprivileged mode. Once in unprivileged mode it is not possible to write to this bit. However interrupts and traps cause a context switch which reloads the status register and this can (and normally does) re-enable privilege.

There is a register sys.int.pend that holds a status bit for each interrupt channel. Bits in sys.int.pend become 1 when an interrupt is requested by an I/O device. They can also be set by software to force a request. Software clears the bit to reset the request. This register is only accessible in privileged mode.

One of the interrupt channels is dedicated to traps. There are many traps, so there is a register sys.trap.pend which identifies which traps are active. Traps can be cleared and set individually in sys.trap.pend. This register is only directly accessible in privileged mode, however there is a trap instruction which can be used by unprivileged code to set any individual bit in sys.trap.pend.

Traps come in two categories. There are error traps that occur when software tries to do something it ought not to, like violating memory access restrictions or executing an unimplemented op code. And there are software traps that are available for any software purpose, such as system calls. Error traps are assigned to bits 31 through 16 in sys.trap.pend, whereas software traps use bits 15 through 0. The assumption is that the trap service routine will use the priority-encode instruction to find the pending trap with the highest priority, and errors should have higher priorities than routine processing.

There is a register sys.int.enable that holds an enable bit for each interrupt channel. sys.int.enable is accessible only in privileged mode. sys.int.enable is cleared on master reset. These enables are grouped in a single register so that kernels which want to do their own interrupt prioritization in software can easily do so.

There are 32 foreground priority levels. 0 is the highest hardware priority level and 31 the lowest. In addition there is priority 32, which is used when no interrupt or trap is active. There is a current priority register (sys.int.priority) which sets the current priority. This register has 6 bits to allow values from 0 through 32. Setting the priority to 0 disables all interrupts. Setting the priority to 32 enables all interrupts. (Of course SR.ie can independently disable all interrupts, regardless of the current priority setting.)

Individual interrupts are not fixed in priority. Instead there is a set of 32 interrupt priority registers, `sys.ip0` through `sys.ip31`, which determine the priority of each interrupt. Note that the priority of an interrupt must not be changed while that particular interrupt is enabled.

Using interrupts

Hardware prioritizing and vectoring can be useful, but software must be able to bypass them easily. Ever since DEC's PDP-11 came out hardware engineers think that automatic prioritization and vectoring of interrupts is the cat's pyjamas. Yet, even when the PDP-11 was new, DEC software engineers were striving to undo these very features.

The PDP-11's multitasking/multiuser operating system was RSX-11. This handled interrupts in two stages. First stage was a single foreground thread in which each job was constrained by fiat to run to completion in 100 microseconds or less. Second stage was any of a number of scheduled high-priority threads. The trouble was that the hardware designers did not envision such an architecture. The RSX-11 programmers were forced to direct each interrupt to a different dedicated block of code which saved context, loaded a register with some identification of the originating interrupt, and then branched to the common entry point. Meanwhile the automatic prioritization was defeated by setting the foreground thread to the highest priority, locking out all interrupts.

I had originally planned for the Gordotron to perform fully automatic context switches. When an interrupt was requested it would cause an automatic switch to the context of the interrupt handler. This has been done before on some processors. However in the end a minimal implementation was not much slower at context switching, and automatically provided great flexibility. So, the final design of the Gordotron-32 provides only the minimal functionality required to respond to an interrupt, as described below. Hardware does not set the new priority level or handle context switching.

The current priority level is determined by the value in `sys.int.priority`. This value ranges from 0 (highest priority) through 32 (lowest priority). When an interrupt is requested, its priority is compared to `sys.int.priority` and if the request priority is numerically equal to or less than `sys.int.priority` the interrupt is granted.

When an interrupt occurs the general registers R0-R31 are copied to a context latch, which has a back-door into the registers so it does not need a memory cycle. Privilege is enabled by setting `SR.privilege`, and at the same time further interrupts are disabled by clearing the `SR.interrupts` bit. The PC is loaded with the value `sys.intsrv`, which points into segment 0. And that is all that hardware does.

The interrupt service code has to initialize any general registers that it needs. The segment table of the interrupted context is still in place, but of course segment 0 is never mapped and never cached. Hence interrupt service code can execute in this mode without disturbing the interrupted context.

Before re-enabling interrupts the interrupt service code must either clear the `sys.int.pend` bit or lower the numeric value in `sys.int.priority`. If it does not do either of these, upon re-enabling interrupts the same interrupt will instantly happen again.

The context latch is mapped to segment 0 as the `sys.ctx` block. The interrupt service code may return directly to the interrupted context with a return instruction, which copies the saved context back into R0 through R31. However in a multitasking system it is often necessary to switch to some other context. In that case the operating system must save the existing contents of the context latch to memory before switching context.

Note that the `sys.intsrv` block is in RAM. The master reset code must set up the `sys.intsrv` block before

enabling interrupts for the first time. Master reset itself jumps to block sys.mstrrst, which is in ROM or flash. Flash memory may be slow but for master reset that is not an issue.

Watchdog

An interrupt is missed when a triggering edge occurs while the corresponding sys.int.pend bit is 1 and the corresponding sys.int.en bit is 1. (Note that the state of SR.ie is disregarded.) Missing any interrupt triggers a watchdog interrupt. Missing a watchdog interrupt triggers a master reset. The watchdog interrupt is itself governed by a bit in sys.int.en like any other interrupt, however unlike the other interrupts once this is enabled it cannot be disabled except by a master reset.

It is normal to set up an on-chip timer to generate interrupts periodically. This ensures that when the system stalls or crashes, failure to serve the timer interrupt eventually triggers a watchdog interrupt.

Gordotron-64

Is it worthwhile supporting 64 priority levels? It seems excessive, but those are famous last words.

Context Switching

The following things must be preserved to save a program's context:

- The contents of the general registers R0-R31.

- The contents of the object buffers, including their lengths.

- The contents of the instruction cache, including its index. (Can be purged without the program noticing, but doing so affects execution time.)

- The segment descriptions.

The Gordotron-32 requires object buffers and the instruction cache to be in high-speed on-chip RAM. These can be placed in different locations for different programs, so it is not necessary to store and retrieve most of this data. However the lengths of object registers are held inside the object manager, and an index to the instruction cache is held inside the instruction cache manager. This internal information must be saved and restored during context switches.

Internal state of the object/vector co-processor (OVPU) is exposed in a 32-word aligned block called `sys.ovpu`.

Internal state of the instruction cache is exposed in a 32-word aligned block called `sys.icache`.

When an interrupt occurs a copy of the general registers is saved in a context latch. The context latch is exposed as a block-aligned 32-word object `sys.ctx`. This same object, when written to, provides a marshaling area for data which you intend to write into the general registers. The actual write from the context latch into R0 through R31 is triggered by a return from interrupt (return) instruction.

Hence to set up a complete context for a foreground program (which has its own dedicated object buffers and instruction cache) the following steps are required:

```
// WARNING: this code must run in segment 0 and with interrupts off
// enter with pointer to context save area in R0
copy 32, R1 // set up constant

// set up the segment registers (while running in segment 0)
ov.get R0, BF0, 0 // get first third of segment descriptors
ov.put sys.seg, BF0; add R1, R0 // put and update pointer
ov.get R0, BF0, 0 // get second third of segment descriptors
ov.put sys.seg+32, BF0; add R1, R0 // put and update pointer
ov.get R0, BF0, 0 // get last third of segment descriptors
ov.put sys.seg+64, BF0; add R1, R0 // put and update pointer

// set up the instruction cache index
ov.get R0, BF0, 0 // get cache internal state 1
ov.put sys.icache.low, BF0; add R1, R0 // put and update pointer
ov.get R0, BF0, 0 // get cache internal state 2
ov.put sys.icache.high, BF0; add R1, R0 // put and update pointer

// set up the register contents
ov.get R0, BF0, 0 // get register contents
ov.put sys.ctx, BF0; add R1, R0 // put and update pointer

// restore registers
return; nil
```

This takes 14 words of instructions, hence it fits in a block. It takes 14 instruction cycles and 12 memory cycles. 6 memory cycles are reading from context store and the rest are writing to segment 0.

Assuming a 1 GHz clock, 8-cycle memory for context store, and 1-cycle memory in segment 0, the foreground task entry time is $14 + (6 * 8) + 6 = 68$ nanoseconds, which is very fast.

To set up the context for a background program (one which shares object buffers and instruction cache) the following steps are required:

```
// WARNING: this code must run in segment 0 and with interrupts off
// enter with pointer to saved object context in R0

// zap the instruction cache index
purge.all; copy 32, R3 // also set up a constant

// copy the saved object buffers to the shared object buffer area
copy sharedArea, R1; copy R3, R2 // set up pointer and counter
loop:
    // This loop could be better optimized. Gets are probably coming from
    // main memory, so slow. Puts are going to high-speed RAM, so fast.
    // Could overlap other processing with the get.
    ov.get R0, BF0, 0; // load 32 words
    ov.put R1, BF0; add R3, R0 // put 32 words; update pointer
    add R3, R1 ; sub 1, R2 // update pointer and count
    br loop; if gt.right

// set up the segment registers (while running in segment 0)
ov.get R0, BF0, 0 // get first third of segment descriptors
ov.put sys.seg, BF0; add R1, R0 // put and update pointer
ov.get R0, BF0, 0 // get second third of segment descriptors
ov.put sys.seg+32, BF0; add R1, R0 // put and update pointer
ov.get R0, BF0, 0 // get last third of segment descriptors
ov.put sys.seg+64, BF0; add R1, R0 // put and update pointer

// set up the register contents
ov.get R0, BF0, 0 // load 32 words of register content
ov.put sys.ctxt, BF0; add R3, R0 // 32-word object

return; nil
```

As coded this takes 15 words, so it fits in a block. It takes 139 instruction cycles and 72 memory cycles. 36 of the memory cycles are reads from the context store and the rest are writes to segment 0. If the clock speed is 1 GHz, the context store is in 8-cycle memory, and high-speed RAM is 1-cycle memory, the background context entry time is $139 + (36 * 8) + 36 = 463$ nanoseconds. These numbers are awesome considering that the context switch horses around 4.5 kilobytes of data.

Part 6:

Instruction Reference

<i>Bit Pattern¹</i>	<i>L?</i>	<i>R?</i>	<i>Group</i>	<i>Functions²</i>	<i>Page</i>
0000 0iss sss0 0010	✓	✓	prefetch	prefetch N prefetch.next	54
0000 0iss sss0 0011	✓	✓	purge	purge N purge.all	55
0000 0100 0001 1100	✓		return	return	56
0000 0iss sss1 1101	✓	✓	trap	trap M	57
0000 0iss sss1 1110	✓		branch	branch M	58
0000 0iss sss1 1111	✓		next	next M	59
0001 xiss sssd dddd		✓	conditional	if.1 M, N if.0 M, N	60
	✓		tandem	(extends right-hand instruction to 64 bit data)	61
001x xiss sssd dddd	✓	✓	calculate	add N, D subtract N, D compare N, N compare.inv N, N	62
01x0 0iss sssd dddd	✓	✓	setbit	setbit.0 M, D setbit.1 M, D	63
01xx xiss sssd dddd	✓	✓	shift	shift.left.0 M, D shift.left.1 M, D shift.left.msb M, D shift.right.msb M, D shift.right.0 M, D shift.right.lsb M, D	64
1000 0iss sssd dddd		✓	encode	encode N, D	65
	✓		co-processor	(uses all 32 bits)	66
10xx xiss sssd dddd	✓	✓	boolean	and.inv N, D copy.inv N, D and N, D xor N, D or N, D copy N, D	67
110x xiss sssd dddd	✓	✓	optional	div.signed N, D div N, D mul.signed N, D mul N, D	69
1111 xiss sssd dddd	✓	✓	memory	load N, D store N, D	70

¹ x – flag; i – immediate; s – source; d – destination

² N – 32 bit constant or register content; M – 5-bit constant or low 5 bits of register content; D – 32-bit destination register

00000 ... 00010 Prefetch Instruction

This instruction warns the instruction cache that a particular code block will or may be used soon. If the code block is not currently in the instruction cache, the cache starts loading it. If the code block is already in the cache, it moves to the front of the queue (most-recently-used).

The source operand is an address anywhere within the code block. The low 5 bits are ignored.

The combination of prefetch with an immediate short operand having the value 0 has a special meaning. It always refers to the block immediately following the current block of instructions. (Note that purge 0 also has a special meaning, but a different one.)

It is forbidden to cache blocks in segment 0, even if the program has privilege to execute from segment 0. If the specified block is in segment 0 the instruction silently does nothing.

There is a queue of pending cache operations. If the queue is full when this instruction is executed, the CPU stalls until a pending operation finishes and a slot becomes available in the queue. The queue is at least 4 deep.

There is no tandem form for this instruction.

This instruction is fail-safe. If the load fails for a trappable reason (for example because access is forbidden) the block silently remains absent from the cache. If and when the program jumps to the block, the CPU will attempt to load the block again. If that in turn fails it will generate a trap.

Flags

None.

Examples

<i>Function</i>	<i>Op Code</i>	<i>I</i>	<i>Src</i>	<i>Op Code</i>
prefetch R6	00000	0	00110	00010
prefetch LBL078	00000	1	11111*	00010
prefetch.next	00000	1	00000	00010

*The immediate value is appended to the instruction.

00000 ... 00011 Purge Instruction

This instruction removes a particular code block from the instruction cache. If the code block is currently in the instruction cache, the cache deletes it, freeing up the storage. But if the code block is not in the cache, the instruction silently does nothing.

The source operand is an address somewhere within the code block. The low 5 bits are ignored when determining which block is intended.

There is a queue of pending cache operations. If the queue is full when this instruction is executed, the CPU stalls until a pending operation finishes and a slot becomes available in the queue. The queue is at least 4 deep.

The combination of purge with a short immediate operand having a value of zero is a special case. It causes the cache for the current process to be purged of code blocks. It also purges any pending cache operations in the queue. (Note that prefetch 0 also has a special meaning, but a different one.)

There is no tandem form for this instruction.

Flags

None.

Examples

<i>Function</i>	<i>Op Code</i>	<i>I</i>	<i>Src</i>	<i>Op Code</i>
purge R6	00000	0	00110	00011
purge LBL078	00000	1	11111*	00011
purge.all	00000	1	00000	00011

*The immediate value is appended to the instruction.

00000 ... 11100 (left) Return Instruction

This instruction performs a return from interrupt or trap. It copies the contents of the dedicated context latch into registers R0 through R31. Since there is only one context latch, and it is written over by any interrupt or trap, code that prepares the latch contents and executes this instruction must run with interrupts disabled.

The source operand is currently not used. To ensure forward compatibility the source and I fields should be set to an immediate short zero. A value of 31 has no special meaning, that is, the instruction is parsed as one that does not have a long immediate mode.

This instruction requires privilege. Any attempt to execute it without privilege causes a privilege violation trap.

This instruction writes to R31 so it can only be used in the left instruction slot. Consequently there can be no tandem form for this instruction, nor would such a thing be useful.

Flags

None.

Examples

<i>Function</i>	<i>Op Code</i>	<i>I</i>	<i>Src</i>	<i>Op Code</i>
return	00000	1	00000	11100

00000 ... 11101 Trap Instruction

This instruction sets one bit in `sys.trap.pend`, forcing a trap to occur at the end of the instruction cycle. The low 5 bits of the source determine which bit is set. This instruction does *not* require privilege.

Traps 16 through 31 are used for error traps, such as permission denied for memory access, however these traps can also be generated using this instruction. Traps 0 through 15 are not used for error traps and may be used for any purpose by software.

This instruction preserves the status of the execution unit in which it runs. This makes it possible to pass status flags into trap service routines.

This instruction is limited to register and short immediate formats. An immediate value of 31 has no special significance.

There is no tandem form for this instruction. However both left and right instructions may be trap instructions, potentially asserting different traps. Error traps may also occur in the same cycle as a trap instruction. It is up to the trap service routine to handle multiple simultaneous traps.

Flags

None.

Examples

<i>Function</i>	<i>Op Code</i>	<i>I</i>	<i>Src</i>	<i>Op Code</i>
trap R6	00000	0	00110	11101
trap 6	00000	1	00110	11101

00000 ... 11110 (left) Branch Instruction

This instruction may only be used in the left-hand slot. It copies the low five bits from the source into the low five bits of the PC. The other bits of the PC are unchanged. The result is a branch which necessarily stays within the same code block, hence does not incur a delay.

This instruction preserves the current state of the left-hand condition flags. Branch PC (specifying the PC as both the source and the destination) acts as a no-op.

This instruction is limited to register and short immediate formats. An immediate value of 31 has no special significance.

Because this instruction only be used in the left-hand slot, there can be no tandem form for this instruction, nor would such a thing be useful.

Flags

None.

Examples

<i>Function</i>	<i>Op Code</i>	<i>I</i>	<i>Src</i>	<i>Op Code</i>
branch R6	00000	0	00110	11110
branch 6	00000	1	00110	11110

00000 ... 11111 (left) Next Instruction

This instruction may only be used in the left-hand slot. It copies the low five bits from the source into the low five bits of the PC. The upper bits of the PC are incremented so that they point to the next sequential block of instructions. The result is a jump that can go to a target anywhere in the next block using only 16 bits.

This instruction preserves the current state of the left-hand condition flags.

This instruction is limited to register and short immediate formats. An immediate value of 31 has no special significance.

Because this instruction only be used in the left-hand slot, there can be no tandem form for this instruction, nor would such a thing be useful.

Flags

None.

Examples

<i>Function</i>	<i>Op Code</i>	<i>I</i>	<i>Src</i>	<i>Op Code</i>
next R6	00000	0	00110	11111
mext 6	00000	1	00110	11111

0001x (right) Conditional Instruction Group

These instructions may only be used in the right-hand slot. They examine one bit (specified by the source) in the destination. If the bit is not in the required state it suppresses the write cycle of the left-hand execution unit, disabling the left instruction. The required state is specified by a flag bit.

These instructions can be used to test any bit in any register. For example, they can be used to test the current bit of the multiplier as part of a multiply step. However they are most commonly used to test bits in the status register, or in a saved copy of the status register.

These instructions are limited to register and short immediate formats. An immediate value of 31 has no special significance.

These instructions have no tandem mode, nor is one necessary.

These instructions cannot directly control a tandem operation, because tandem operations consume both instructions in the pair. To conditionally execute a tandem operation you have to conditionally branch over it.

Flags

F1: state

This flag supplies the reference state. To meet the condition (causing the left-hand instruction to be executed) the bit under test must match the reference state.

Examples

<i>Function</i>	<i>Op Code</i>	<i>State</i>	<i>I</i>	<i>Src</i>	<i>Dst</i>
if.0 R2, R6	0001	0	0	00110	00010
if.0 6, R6	0001	0	1	00110	00010
if.1 R2, R6	0001	1	0	00110	00010
if.eq.left	0001	1	1	01111	11110
if.eq.right	0001	1	1	00111	11110
if.lt.left	0001	1	1	01011	11110
if.le.left	0001	1	1	01010	11110
if.ne.left	0001	0	1	01111	11110
if.ge.left	0001	0	1	01011	11110
if.gt.left	0001	0	1	01010	11110
skip	0001	1	1	10000	11110
nil	0001	0	1	10000	11110

0001x (left) Tandem Instruction Group

These instructions may only be used in the left-hand slot. They cause the left-hand processing unit to slave itself to the right-hand unit. They may also modify the behaviour of the right-hand unit. For example, during a circular tandem shift the left-hand unit shifts in up to 31 bits from the right-hand unit, while the right-hand unit shifts in the same number of bits from the left-hand unit.

When the instruction supplied to the right-hand unit is in the **memory** or **shift** group, the left-hand unit does not supply a source operand. In such cases the left-hand immediate flag must be 0 and the left-hand source field must be all zeros. When the instruction supplied to the right-hand unit is in the **coprocessor** group, the behaviour is determined by the coprocessor. In all other cases this instruction supplies both source and destination operands and can use any of register, short immediate, or long immediate format.

Tandem Boolean operations (including copy) produce the same result as two separate 32-bit operations, except that the left result flags reflect the entire 64-bit result.

Flags

F1: x

The x (extended) flag may alter the tandem operation. Many tandem operations ignore this flag. See the individual instruction descriptions for more information. If the description does not mention the tandem x flag then it is ignored and, for forward compatibility, must be set to zero.

Examples

In assembler notation tandem operations are identified by paired left:right operands. In these examples each 64-bit operand is in a sequential pair of registers but that is only a convention.

```
add R2:R3, R6:R7           // 64-bit add
add HIWORD:LOWORD, R6:R7    // 64-bit add immediate
compare R2:R3, R6:R7        // 64-bit compare
test R6:R7                  // 64-bit test
store ADDR:ADDR+1, R6:R7    // 64-bit store
load BF17.X:Y, R6:R9         // load 2 different properties from an object
shift.left.c PLACES, R6:R7   // 64-bit shift by up to 31 places

// 128-bit add takes two cycles
add R2:R3, R6:R7
add R4:R5, R8:R9

// 96-bit add takes 1.5 cycles
add R2, R6; ...
add.c R3:R4, R7:R8
```

001xx Calculate Instruction Group

These instructions calculate using an integer arithmetic unit. The operation may be an add or a subtract.

Adds and subtracts with the PC specified as the destination perform relative branches.

Note that there is no instruction that performs a negate in place. To negate a register in place you have to first complement it in place and then increment it in place.

In tandem mode these instructions perform 64 bit calculations. If the x flag on the tandem instruction is set the calculation also takes in the left carry bit.

Flags

F1: subtract

When this flag is 1 the source is subtracted from the destination.

When this flag is 0 the source value is added to the destination.

F2: compare

When this flag is 0 the operation is a normal add or subtract. Both the destination and the status flags are updated.

When this flag is 1 the operation is a compare. The destination is not modified but the status flags are updated as usual.

Examples

<i>Function</i>	<i>Op Code</i>	<i>Subtract</i>	<i>Compare</i>	<i>I</i>	<i>Src</i>	<i>Dst</i>
add R6, R2	001	0	0	0	00110	00010
add 6, R2	001	0	0	1	00110	00010
add 9999, R2	001	0	0	1	11111*	00010
increment R2	001	0	0	1	00001	00010
subtract R6, R2	001	1	0	0	00110	00010
decrement R2	001	1	0	1	00001	00010
compare R6, R2	001	1	1	0	00110	00010
compare -6, R2	001	0	1	1	00110	00010
test R2	001	0	1	1	00000	00010
jump (. - 6)	001	1	0	1	00110	11111
jump (. - target)	001	1	0	1	11111**	11111

*The value 9999 is appended to the instruction.

**The value of the current address minus the target address is appended to the instruction.

Assembler expansion

add.c R2:R3, R6:R7 → tandem.x R2, R6; add R3, R7

01x00 Setbit Instruction Group

These instructions set or clear individual bits in the destination. The source supplies the bit number.

These instructions are limited to register and short immediate formats. An immediate value of 31 has no special significance.

These instructions do not operate in tandem mode.

Flags

F0: value

The selected bit is set to the value supplied by this bit.

Examples

<i>Function</i>	<i>Op Code</i>	<i>Value</i>	<i>Op Code</i>	<i>I</i>	<i>Src</i>	<i>Dst</i>
setbit.0 R6, R2	01	0	00	0	00110	00010
setbit.0 6, R2	01	0	00	1	00110	00010
setbit.1 R6, R2	01	1	00	0	00110	00010
setbit.1 6, R2	01	1	00	1	00110	00010
setbit.1 z.left	01	1	00	1	01111	11110
setbit.1 z.left, R2	01	1	00	1	01111	00010

01xxx Shift Instruction Group

These instructions shift the destination by from 0 to 31 places.

These instructions are limited to register and short immediate formats. An immediate value of 31 has no special significance.

In tandem mode the left-hand unit does not use a source operand. It shifts the same number of places that the right-hand unit shifts. In this mode the destination is treated as a single 64-bit number, unless the tandem op code is tandem.x and the operation is a circular shift, in which case the carry bit in the status register plus the operand together are treated as a 65-bit operand.

Flags

F3: direction

When this bit is 0 the shift direction is left.

When this bit is 1 the shift direction is right.

F1, F2: in

These two bits control what value is shifted into vacated bit positions as follows:

01: Circular shift. Bits that shift out at one end shift in again on the other end. For a right shift the LSB is shifted in, whereas for a left shift the MSB is shifted in.

10: Logical shift. All bits shifted into vacated bit positions are zeros.

11: For a left shift, all bits shifted into vacated bit positions are ones (typically used to generate a mask). For a right shift they are copies of the original MSB (arithmetic shift).

Examples

<i>Function</i>	<i>Op Code</i>	<i>Direction</i>	<i>In</i>	<i>I</i>	<i>Src</i>	<i>Dst</i>
shift.left.msb R6, R2	01	0	01	0	00110	00010
shift.left.msb 6, R2	01	0	01	1	00110	00010
shift.left.0 R6, R2	01	0	10	0	00110	00010
shift.left.1 R6, R2	01	0	11	0	00110	00010
shift.right.lsb R6, R2	01	1	01	0	00110	00010
shift.right.0 R6, R2	01	1	10	0	00110	00010
shift.right.msb R6, R2	01	1	11	0	00110	00010

Assembler expansion

```
shift.right.c R6, R2:R3 → tandem.x 0, R2; shift.right.lsb R6, R3
```


10000 (right) Encode Instruction

This instruction may only be used in the right-hand slot. It sets the destination to the bit number of the most significant one bit in the value of the source.

When the source is all zeros this instruction asserts the overflow status bit associated with the processor, and writes all zeros to the source. However all zeros is also the value associated with a valid source value of hex 00000001. The overflow flag distinguishes between the two values.

In tandem mode the two processors act together to priority-encode the 64-bit source, yielding a six-bit number. In this mode the left destination register specification is not used and must be set to all zeros. If the 64-bit source is all zeros the overflow flag of the left processor is asserted.

This instruction has long and short immediate forms, but they are redundant because the assembler can do the encode operation at assembly time. Normally this instruction is used with a source register either to find the first 1 bit in a bitmap or as the first step in normalizing a floating-point mantissa.

Flags

None.

Examples

<i>Function</i>	<i>Op Code</i>	<i>I</i>	<i>Src</i>	<i>Dst</i>
encode R6, R2	10000	0	00110	00010

10000 (left) Co-processor Instruction

This instruction may only be used in the left-hand slot. It consumes all 32 bits of the instruction word. It sends a particular command, plus operands, to a particular co-processor. The operands include two source operands, either of which may be an immediate value, and a destination.

The format of the instruction word is show below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	i	src.left					co-processor select					co-processor command					i	src.right					dst				

The co-processor is selected by the five-bit value in bits 16 through 20. The command for the processor is selected by the five-bit value in bits 11 through 15. The meaning of the command is specific to the co-processor.

The distinction between the co-processor select and command fields is not absolute. A co-processor that needs more than 32 commands might claim two select values. Two co-processors that together need no more than 32 commands might share a single select value. However it is convenient to visualize these as separate 5-bit fields.

The co-processor is supplied with a copy of the operand fields (src.left, src.right and dst) and source indirect bits (i.left and i.right). It may, at its choice, interpret any of the operand fields in some non-standard way. However it can also use the resolved 32-bit operand values as ordinary instructions do, and it can update the destination register as ordinary instructions do. It is *not* a tandem instruction, hence it is not expected to treat the left and right source operands as halves of a 64-bit quantity. Instead we supply three operands because many co-processor instructions are rather high-level and require three operands.

In many cases co-processors operate in parallel with the CPU and can not produce immediate status updates. It is therefore normal for a co-processor to have its own internal status register. Co-processors can also cause traps using the normal trap mechanism, but if the error detection is delayed the co-processor must generate an interrupt rather than a trap. During interrupt processing the CPU syncs with all co-processors, waiting if necessary until they are all idle or in a restartable state.

Flags

None built in. The co-processor may have a status register which may include result flags. For synchronous operations it may also set the result flags in the SR.

Examples

<i>Function</i>	<i>Op Code</i>	<i>I.left</i>	<i>Src.left</i>	<i>Select</i>	<i>Cmd</i>	<i>I.right</i>	<i>Src.right</i>	<i>Dst</i>
coproc 7, 26, R6, 17, R2	10000	0	00110	00111	11010	1	10001	00010

Assembler expansion

It is usual to define a class for each co-processor. For example, if there is a video processing co-processor, and it implements a command to copy its status register into a destination register, the assembler syntax of the command might be `video.getstatus R2`.

10xxx Boolean Instruction Group

These instructions compute each result bit as a Boolean function of the source and destination bits at the same position in the word. The function is determined by the flag bits.

One combination of flag bits results simply in copying the source to the destination. This is the normal way to perform copies. If the destination is the PC then the copy performs a jump to an absolute address.

Another combination of flag bits copies the ones-complement of the source to the destination. This has two common additional uses. One is complementing a register in place, by specifying the same source and destination. The other is initializing a register to a small negative value, using the short immediate format.

In tandem mode the boolean operation is performed between two 64-bit values. This is not quite the same as two separate 32-bit boolean operations, because in tandem mode the left status flags reflect the whole 64 bits.

Flags

F3, F2, F1:

001: and ($S_n \cdot D_n \rightarrow D_n$)

010: copy inverted ($\sim S_n \rightarrow D_n$)

011: and inverted ($\sim S_n \cdot D_n \rightarrow D_n$)

101: copy ($S_n \rightarrow D_n$)

110: or ($S_n + D_n \rightarrow D_n$)

111: xor ($(\sim S_n \cdot D_n) + (S_n \cdot \sim D_n) \rightarrow D_n$)

These encodings were chosen for fast and easy decoding. For more information see page [91](#).

Examples

<i>Function</i>	<i>Op Code</i>	<i>F2, F1, F0</i>	<i>I</i>	<i>Src</i>	<i>Dst</i>
and R6, R2	10	001	0	00110	00010
and 6, R2	10	001	1	00110	00010
and 9999, R2	10	001	1	11111*	00010
copy.inv R6, R2	10	010	0	00110	00010
copy -7, R2	10	010	0	00110	00010
invert R2	10	010	0	00010	00010
and.inv R6, R2	10	011	0	00110	00010
copy R6, R2	10	101	0	00110	00010
copy 6, R2	10	101	0	00110	00010
copy 9999, R2	10	101	1	11111*	00010
jump R6	10	101	0	00110	11111
jump target	10	101	1	11111*	11111
or R6, R2	10	110	0	00110	00010
xor R6, R2	10	111	0	00110	00010

*The immediate value is appended to the instruction.

110xx Optional Instruction Group

This group of instructions perform multiply and divide. Low-end MCUs often do not need these instructions, so this group is optional. It is expected that on such machines the operating environment will catch unimplemented op code traps for these instructions and emulate them in software.

These instructions operate on a pair or 32-bit integer operands, yielding a 32-bit result. The operands and result may be signed or unsigned. If the result does not fit into 32 bits the overflow flag for the execution unit is set.

Even when implemented in hardware, instructions in this group may take more than one cycle to execute. If so both execution units stall until the multiply or divide is complete.

This instruction may be used in tandem mode, in which case it operates on two 64-bit operands, yielding a 64-bit result. Tandem operations may take more cycles to execute than non-tandem operations.

Flags

F1: Divide

When this flag is set the operation is a divide. When it is cleared the operation is a multiply.

F2: Signed

When this flag is set the operands and result are signed. When it is cleared they are unsigned.

Examples

<i>Function</i>	<i>Op Code</i>	<i>Divide</i>	<i>Signed</i>	<i>I</i>	<i>Src</i>	<i>Dst</i>
mul R6, R2	1110	0	0	0	00110	00010
mul 9999, R2	1110	0	0	1	11111*	00010
mul.signed R6, R2	1110	0	1	0	00110	00010
div R6, R2	1110	1	0	0	00110	00010
div 9999, R2	1110	1	0	1	11111*	00010
div.signed R6, R2	1110	1	1	0	00110	00010

Usage

```
// Multiply two 32-bit unsigned numbers to generate a 64-bit result
// Multiplier is in R4, multiplicand in R19 and result goes to R22:R19.
copy 0, R22; ... // initialize upper half of destination to zero
mul 0:R4, R22:R19 // upper half of source is immediate short zero
```

1111x Memory Instruction Group

This group of instructions transfer data between the CPU and memory or an addressable device.

The load instruction copies a word from memory or an addressable device to the destination register. The store instruction copies a word from the destination register to memory or an addressable device, leaving the destination register unchanged.

These instructions are synchronous. The CPU stalls until the operation is complete. The instruction may also have to wait until a pending operation completes before it can start its own operation. If the addressed memory block is in a virtual partition, and the block is not resident, and a software page fault is generated, then the instruction aborts and is re-executed when the context is re-entered.

Memory segments have permission and size restrictions. In addition segment 0 has permissions that vary from block to block. Some segments, and some blocks in segment 0, require privilege to write or to access. If the requested access violates these restrictions the instruction generates a memory access trap.

There is only one memory channel, so only one of the instructions in a pair can be from this group. However that one can be in either the left or right slot. An attempt to execute two instructions from this group in the same cycle generates an unsupported operation trap.

This instruction may operate in tandem mode. In tandem mode only the low 5 bits of the right-hand source operand are used, so that both units address the same block (or data object in an object buffer) but can operate on different words within the block (or data object). Because both units are accessing the same block they can operate simultaneously. If they both write to the same word the result is the bitwise AND of the two data words.

When the operation is a store in tandem mode, and the tandem x flag is set, the memory location referenced by the left-hand operand is a semaphore. Both memory locations are read, and if the semaphore is non-zero the semaphore is already in use. In this case the store fails, setting the left overflow flag. However if the current semaphore value is zero then the supplied values are written into the memory locations and the left overflow flag is cleared. This is all done with a single read-modify-write cycle so that no race condition can exist (i.e. it is thread-safe).

Flags

F1: store

When this flag is 0 the operation is a load, that is, the contents of the memory location or external register are copied into the destination register. The instruction fails, generating a trap and leaving the destination unchanged, if the specified location is not readable.

When this flag is 1 the operation is a store, that is, the contents of the destination register are copied into the memory location or external register. The destination register is not altered. The instruction fails, generating a trap and leaving memory unchanged, if the specified location is not writable.

Examples

<i>Function</i>	<i>Op Code</i>	<i>Store</i>	<i>Object</i>	<i>I</i>	<i>Src</i>	<i>Dst</i>
load R6, R2	1111	0	0	0	00110	00010
load memloc, R2	1111	0	0	1	11111*	00010
store R6, R2	1111	1	0	0	00110	00010
store memloc, R2	1111	1	0	1	11111*	00010

*The address of memloc is appended to the instruction.

Usage

```
// read a register
load PORT7, R20; ...      // may stall at option of device
copy R20, R21; ...        // data is valid now

// read from an array
// compute address in R10
load R10, R0; ...         // main memory access stalls

// tandem load two properties from an object in an object buffer
// X and Y are defined as offsets of properties within the object
load BF5.X:Y, R1:R8        // expands to tandem 160+X, R1; load Y, R8
```

Part 7:

Extended Instructions (64-bit only)

<i>Bit Pattern¹</i>	<i>L?</i>	<i>C?</i>	<i>R?</i>	<i>Instruction Group</i>	<i>Functions²</i>
1 00xx xiss ssss SSSS SSdd dddd		✓		floating point	(Not defined.)
101 00xi ssss ssdd dddd	✓		✓	tandem	(Extends precision of instruction to its right.)
1 0100 xiss ssss SSSS SSdd dddd		✓		tandem	(Extends precision of instruction to its right.)
101 01xi ssss ssdd dddd	✓		✓	conditional	if.0 M, N if.1 M, N
101 01xiss ssss SSSS SS11 1111		✓		conditional	if.0 M, N if.1 M, N

¹ SSSSSS – second source register for three-operand instruction.

² N – 32 bit constant or register content; M – 5-bit constant or low 5 bits of register content.

100xxx Floating-Point Instruction Group

Not defined.

10100x Tandem Instruction Group

This tandem instruction group has its own distinct op code range. It can therefore be used in any position. It causes the execution unit to slave itself for this cycle to the execution unit to its right, with the rightmost unit slaving itself to the leftmost unit.

When the center execution unit is part of a tandem instruction, it remains a three-operand unit even though it is operating in tandem with two-operand units. Therefore the center unit can still specify two different registers for the second source operand and the destination.

It is acceptable for two units to slave themselves to the third unit. For example if the left and center instructions both use this op code, and the right hand instruction is an add, the result is to add two 192-bit operands. It is forbidden for all three units to execute instructions in this group at the same time, because there would be no non-tandem instruction for them to slave themselves to.

10101x Conditional Instruction Group

This conditional instruction group has its own distinct op code range. It can therefore be used in any position. It causes the execution unit to test a condition and, if the condition is not met, inhibit the unit immediately to its left, with the leftmost unit inhibiting the rightmost unit. By contrast the 5-bit conditional instructions, when executed by the rightmost unit, conditionally inhibit *both* the center and left units.

When used in the center position, only the two sources are used. The destination field is not used. The destination field must be set to all ones.

It is forbidden for all three units to execute instructions in this group at the same time. There would be no processing to inhibit.

Part 8:

OVPU Command Reference

<i>Op Codes</i>	<i>Group</i>	<i>Commands</i>	<i>Page</i>
0, 1	register	read, write	77
2 – 7	memory access	get, put, getutf8, pututf8, getitem, setitem	78
8 – 14	vector-to-vector	add, sub, mul, div, and, or, xor	80
15 – 18	shift	shift.right, shift.left, rotate.right, rotate.left	81
19, 20	single-vector	fill, resize	82
21 – 26	selection set	value, compare, subset, saveset, include, exclude	83

Register command group

These command write to and read from the accessible registers.

The commands are summarized below.

<i>Op code</i>	<i>Assembler</i>	<i>Group</i>	<i>Operation</i>
0	ov.read regnum, dst	status	Copy contents of OVPU register to destination
1	ov.write regnum, src	status	Copy contents of left source to OVPU register

The OVPU register number is the left source operand. It must be a value as listed in the table below.

<i>Register Number</i>	<i>Name</i>	<i>Access</i>	<i>Description</i>
0	ID	Read-only	Code for type of processor (= 1)
1	Status	Privileged write	Status flags
2	Mode	Read/write	Settings to alter commands
3	Left Index	Read-only	Position of the leftmost item in the selection set
4	Right Index	Read-only	Position of the rightmost item in the selection set

For ov.read the data is written to the register (R0-R30) specified by the destination operand. For ov.write the data is obtained from the right source operand.

Memory-access command group

The get and put commands are used to move objects or parts of objects in and out of buffers or aligned memory blocks. The load and store commands are used to move words from memory into and out of registers.

These commands ignore the selection set fields of the mode register, because they are intended for use with objects rather than vectors (although the objects may in fact be vectors). They also ignore the overflow field because they cannot encounter numeric overflow conditions. Some of them use the item size field but the others ignore it.

The memory-access commands are listed below.

<i>Op code</i>	<i>Assembler</i>	<i>Group</i>	<i>Operation</i>
2	ov.get address, count, buffer	memory	Copy words from memory to an object buffer
3	ov.put address, count, buffer	memory	Copy words from an object buffer to memory
4	ov.getutf8 block, index, buffer	memory	Copy a string from memory to an object buffer, transcoding from UTF-8 to UCS-4
5	ov.pututf8 block, index, buffer	memory	Copy a string from an object buffer to memory, transcoding from UCS-4 to UTF-8
6	ov.getitem block, index, dst	memory	Copy item at index in object buffer to destination register
7	ov.setitem block, index, dst	memory	Copy item from destination register to object buffer at index
internal	load store address:index2, dst1:dst2	memory	Address memory or object buffer on behalf of a CPU load or store instruction

In the above table, `address` refers to a concatenation of a 27-bit block selector and a 5-bit index.

The `ov.get` operation left-shifts the data to align it with the block and fills the remaining words in the block (past the count) with zeros. The `ov.put` operation right-shifts the data to align it with the memory address and writes only up to the word count.

For these commands the object buffer is specified by the destination field, the memory address by the left source field, and the count by the right source field. The memory region is not necessarily aligned and may even span blocks. A count of 0 means a 32-word long object. Other count values are the length of the object in words. If the length specified extends past the memory block containing the start address, the OVPU executes a second memory cycle to complete the transfer.

The `ov.getutf8` and `ov.pututf8` commands copy and translate a string between two commonly used text representations. UTF-8 is used in communications, in interchange formats such as XML, and often in storage because it is relatively compact. UCS-4 (32 bits per character) is used inside programs because it is easy to handle.

Both of these instructions assume that the string is stored unaligned in memory as UTF-8 but in an object buffer as UCS-4. The `ov.getutf8` command copies and translates from unaligned memory to an object buffer. The `ov.pututf8` command copies and translate from an object buffer to unaligned

memory. Either command stops when it reaches either the end of the buffer or a null terminator. The UTF-8 string in memory may cross block boundaries. If necessary the OVPU executes a second memory cycle to complete the operation.

UTF-8 uses variable-length encoding for characters. As specified for Unicode each character takes from 1 to 4 bytes of memory. Consequently the index variable has a range from 0 to 127 to reflect the fact that, when operating on a substring, you may be starting at any byte boundary.

This operation does not respect the selection set. However at the end of the operation the selection set is set to reflect how many bytes were used from the UTF-8 string, starting at the supplied block and index. The program can read the right index register to find out how many UTF-8 bytes were successfully written or read. To determine how many UCS-4 characters were processed the program has to look for the null terminator in the buffer (see `ov.value` below). If there is no null terminator then exactly 32 UCS-4 characters were processed, and the operation terminated without processing a null terminator because the buffer was exhausted.

These two instructions implement full UTF-8 and Unicode validation. This is necessary because invalid codes can be and have been used to bypass security. For example, in applications where only relative URLs are allowed, absolute URLs have been slipped past code that was checking for “/” with a naïve substring search. If an invalid code is detected the operation stops early, with the right index register indicating how many bytes were successfully processed before stopping. In addition the status register is set to indicate a data error. If the UTIE bit is set in the mode register the error also generates an interrupt. (An interrupt rather than a trap because these are asynchronous operations. Hence the interrupt may come long after the instruction was issued.)

These operations are rather slow, as hardware operations go. For a practical gate count they will use sequential logic and probably take one cycle per character, although 2 per cycle might be doable. Nonetheless this is much faster than software can manage. For data center applications this is a significant win.

The `ov.getitem` and `ov.setitem` commands allow getting or setting one item within a vector. All item sizes are supported. Note that the index value can be up to 127 because the items may be as small as 8 bits. Hence the buffer and index must be specified separately, not combined into a single address as the load and store commands do. For both commands the left source specifies the buffer or aligned block of memory, the right source specifies the index, and the destination register either receives or supplies the data.

The CPU's load and store instructions do not use the co-processor op code but nevertheless part of their functionality is implemented by this co-processor. The OVPU monitors the address bus. When the address bus specifies a block selector value of 0 through 31 the OVPU copies the selected words of the appropriate object buffer to the data bus or loads them from the data bus as appropriate.

CPU loads and stores can take place while the OVPU is busy processing a previously issued command. CPU loads and stores take precedence over OVPU operations, unless the load or store cannot be performed yet because the OVPU has not yet completed an operation using the same object buffer or block of memory. In that case the OVPU stalls the CPU until it is safe for the CPU load or store to take place. For example, if the OVPU is working on `ov.and 0xAB0, 0xAC8, 15`, but has not yet fetched the second source operand, and the CPU attempts to write to the block at `0xAC8`, the OVPU stalls the CPU until the OVPU has finished loading the existing content from `0xAC8`.

Vector-to-vector command group

Vector-to-vector commands perform single-instruction multiple-data (SIMD) operations between two operands, each of which is a 32-word vector contained in an aligned block of memory or buffer. The content of each block is treated as a vector of independent values. An operation is performed independently upon each pair of values from the source vectors, and for every operation but `ov.compare` the result is written to the destination vector.

All of the vector-to-vector commands are three-operand instructions. They can be modified by the application of the selection set, by specifying the item size, and for some operations by specifying an overflow behavior.

The vector operations are listed below.

<i>Op code</i>	<i>Assembler</i>	<i>Group</i>	<i>Operation</i>
8	<code>ov.add block1, block2, buffer</code>	vec-vec	Add items
9	<code>ov.sub block1, block2, buffer</code>	vec-vec	Subtract items
10	<code>ov.mul block1, block2, buffer</code>	vec-vec	Multiply items
11	<code>ov.div block1, block2, buffer</code>	vec-vec	Divide items
12	<code>ov.and block1, block2, buffer</code>	vec-vec	Boolean AND items
13	<code>ov.or block1, block2, buffer</code>	vec-vec	Boolean OR items
14	<code>ov.xor block1, block2, buffer</code>	vec-vec	Boolean XOR items

The left source supplies a block selector for the first source vector, the right source field supplies a block selector for the second source vector, and the destination field specifies an object buffer for the result.

The `ov.mul` and `ov.div` commands require multiple cycles to perform the computation. To keep the gate count reasonable the multiply or divide is done sequentially, perhaps using Booth's algorithm.

Shift command group

These commands copy and shift a vector. The left source supplies a count. The right source supplies a block selector for the vector. The destination field selects an object buffer.

<i>Op code</i>	<i>Assembler</i>	<i>Group</i>	<i>Operation</i>
15	ov.shift.left block, count, buffer	vector	Shift items to the left, shifting in zeros
16	ov.shift.right block, count, buffer	vector	Shift items to the right, shifting in zeros
17	ov.rotate.left block, count, buffer	vector	Rotate items to the left, circular
18	ov.rotate.right block, count, buffer	vector	Rotate items to the right, circular

These commands shift the items in a vector left or right. The count is in items, not words, so it can be up to 127 for 8-bit items. However it must not be more than 63 for 16-bit items or 31 for 32-bit items. (If you want to fill a buffer completely with zeros use ov.fill, below.) The minimum shift count is 0.

The left source supplies a block selector for the source vector. The right source supplies a count for the shift. The destination field specifies an object buffer to which the result is written.

These commands are affected by the selection set and item size fields in the mode register. They disregard the overflow field because they cannot generate overflows.

The assembler defines ov.copy as a synonym for ov.shift with a shift count of zero.

Single-vector command group

These commands operate upon a single vector.

<i>Op code</i>	<i>Assembler</i>	<i>Group</i>	<i>Operation</i>
19	ov.fill pattern, buffer	vector	Fill with a pattern
20	ov.resize newsize, buffer	vector	Resize items, throwing away extra items and padding with zeros as necessary

For these instructions the pattern or new size is supplied by the left source and the destination selects an object buffer. Both of these commands respond to the item size and subset fields in the mode register. The ov.resize command also responds to the overflow field in the mode register when the new size is smaller than the current item size.

Selection set command group

These commands operate upon the selection set.

<i>Op code</i>	<i>Assembler</i>	<i>Group</i>	<i>Operation</i>
21	ov.value block, value, condition	Selection set	Select a subset by comparing items to a value
22	ov.compare block1, block2, condition	Selection set	Generate a subset by comparing items numerically
23	ov.subset start, end	Selection set	Set selection set to a contiguous subset
24	ov.saveset buffer	Selection set	Save selection set as a 128-byte vector
25	ov.include index	Selection set	Add one item to the selection set
26	ov.exclude index	Selection set	Remove one item from the selection set

The ov.compare command requires one machine cycle plus two memory cycles. The ov.saveset and ov.value commands require one machine cycle plus one memory cycle. The others only require one machine cycle because they are purely internal operations.

The ov.value command compares items in the source vector to a value. The ov.compare command compares items in the first source vector to corresponding items in the second source vector. In both cases corresponding bits in the selection set are modified according to the results of the individual comparisons, in accordance with the condition operand. The condition operand is coded as follows.

3	2	1	0
Reverse	Signed	Comparison	

The comparison field is coded as follows.

<i>Code</i>	<i>Comparison</i>	<i>Mnemonic</i>	<i>Reversed Comparison</i>	<i>Mnemonic</i>
0	src1 equal src2	eq	src1 not equal src2	ne
1	src1 less than src2	lt	src1 greater than or equal to src2	ge
2	src1 less than or equal to src2	le	src1 greater than or equal to src2	gt
3	all items	all	no items	none

The number line for these conditions is determined by the signed bit. If it is a zero then an unsigned number line is used (for eight-bit items, 0 through 255). If it is a one then a signed number line is used (for eight-bit items, -128 through +127).

There are 128 bits in the selection set register. Commands that modify the selection set change selection

bits individually if the item size is 8 bits, but in pairs if the item size is 16 bits and in quads if the item size is 32 bits. However, when any command responds to the selection set, inclusion is determined by the OR of the bits for that item, so for example if the item size is 32 bits it is included if any bit in its quad is 1.

Commands that modify the selection set can themselves be limited to a subset of the items. Each item that is tested and meets the condition is added to the selection set. The others that are tested and fail to meet the condition are removed from the selection set. Membership of the items that are not tested is not affected. Remembering that the subset can be the selection set or the inverse of the selection set, you can use successive commands either to build up the selection set (union) or winnow it down (intersection).

The `ov.saveset` command saves the selection set as a vector of 128 bytes (not bits), with each byte containing either 0 (excluded) or 1 (included). Executing `ov.value ne, buffer, 8` on the saved vector will restore the selection set.

Part 9: Implementation

Physical Memory Segment

The following table shows the map for segment 0, the physical memory segment.

<i>Block Selector</i>	<i>Name</i>	<i>Access</i>	<i>Description</i>
0 ... 31	sys.buf0...31	Read/Write	Current set of object buffers, when block selector is resolved by the OVPU
32	sys.ctxt	Privileged and buffered	Saved (or about to be restored) R0...R31
33, 34	sys.ic0...ic32, sys.cbb, sys.cc	Privileged	Interrupt context control
35	sys.int, sys.trap	Privileged	Interrupt and trap control
36	sys.ip0 ... 31	Privileged	Priority levels for each hardware interrupt
37	sys.ovpu	Privileged	OVPU internal state
38	sys.icache	Privileged	Instruction cache internal state
39 ... 63	N/A	N/A	Reserved for other devices
64 ... 84	sys.part	Privileged	Partition table
85 ... 87	sys.seg	Privileged	Segment table
87 ... 32766	N/A	N/A	Reserved for other devices
32767	sys.mstrst	Privileged	Entry point for master reset (ROM or flash)
32768	sys.intsrv	Privileged	Entry point for interrupt service (RAM)
32769+	N/A	Privileged	General-purpose high speed RAM

Traps

The following chart shows the hardware traps in sys.trap.pend.

<i>Bit</i>	<i>Name</i>	<i>Description</i>
31	sys.trap.pend.operation	Illegal or unimplemented instruction or combination of instructions.
30	sys.trap.pend.privilege	Operation denied for lack of privilege.
29	sys.trap.pend.access	Access denied by the memory management unit.
28	sys.trap.pend.buffer	Attempt to use data in an object buffer after the get for that buffer failed.
16	sys.trap.pend.overflow	Overflow trap. An instruction set the SR.v.left flag while SR.tv.left was true and/or set the SR.v.right flag while SR.tv.right was true.

Decoding the word-enable lines for the OVPU

We need a decoder that, instead of activating only one output for each input combination, activates first one output, then two, then three, and so on. I don't know what such a decoder is called. If we reduce it to a two-line to four-line decoder, the truth table looks like this.

<i>Select 1 ... 0</i>	<i>Decode 3 ... 0</i>
00	0001
01	0011
10	0111
11	1111

But we need to be able to gang together multiple such decoders to make a bigger decoder. There is a straightforward way to do this. Add two inputs to each decoder, as follows.

<i>After (H)</i>	<i>Before (L)</i>	<i>Select 1 ... 0 (H)</i>	<i>Decode 3 ... 0 (H)</i>
L	H	LL	LLLH
L	H	LH	LLHH
L	H	HL	LHHH
L	H	HH	HHHH
H	H	XX	HHHH
L	L	XX	LLLL

The combination of After and Before both being asserted is forbidden. The gating equations are:

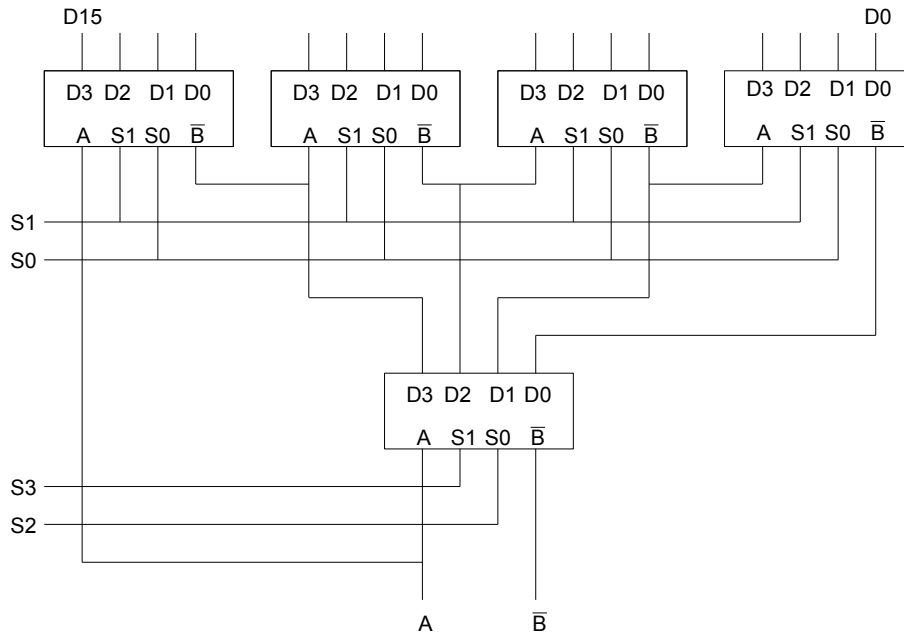
```
Decode[0] (H) = Before
Decode[1] (H) = (Before & (Select[0] | Select[1])) | (After)
Decode[2] (H) = (Before & Select[1]) | (After)
Decode[3] (H) = (Before & Select[1] & Select[0]) | (After)
```

After reduction this takes 7 NAND gates and 3 inverters, as follows.

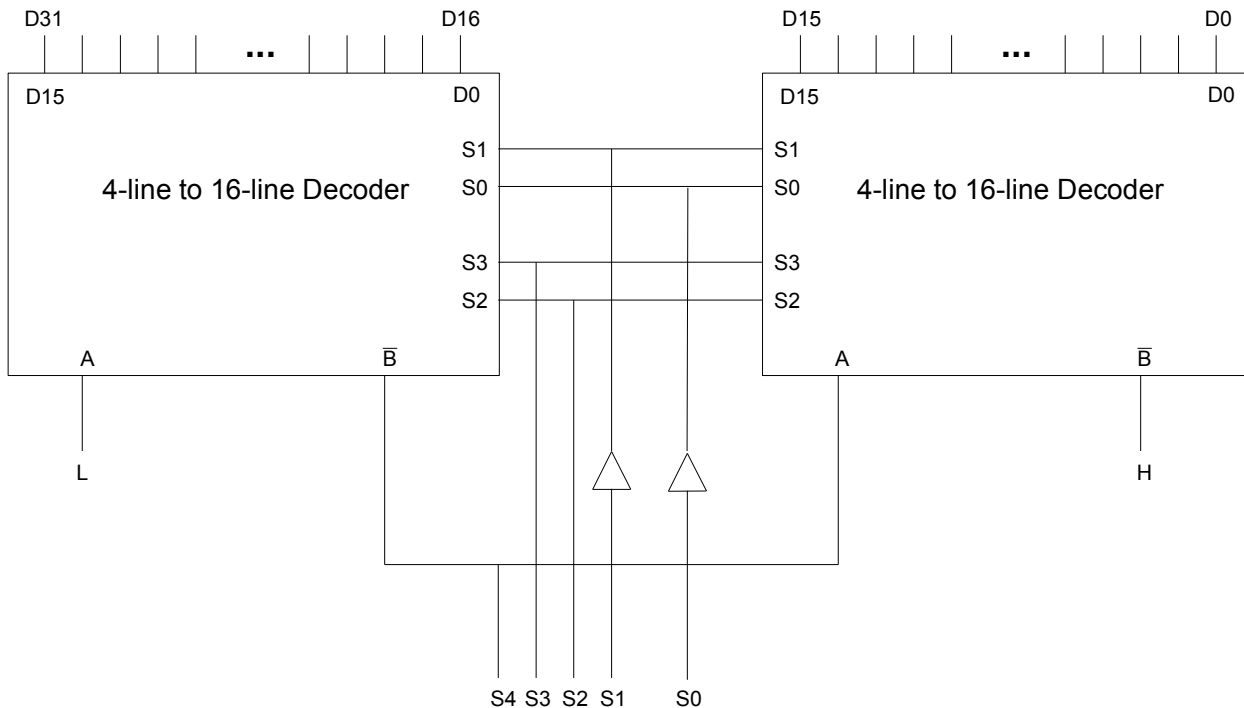
```
Decode[0] (H) = ~(~(Before))
Decode[1] (H) = ~(Before & Select[0]) & ~(Before & Select[1]) & ~(After)
Decode[2] (H) = ~(Before & Select[1]) & ~(After)
Decode[3] (H) = ~(Before & Select[1] & Select[0]) & ~(After)
```

The stage delay is equal to two gate delays.

Now we can cascade two levels of such decoders, as follows, to make a 4-line to 16-line decoder.



But we need 5-line to 32-line. Fortunately, we can expand to that size without adding more levels of decoding. To avoid fan-out problems we do have to add some buffering for S0 and S1, but it doesn't add any gate delays because it is in parallel with the first-stage decoder.



Now we can control the word enable lines using two such decoders, one for the start index and the other for the end index. A word enable line is enabled if its corresponding start bit is asserted but its corresponding stop bit is not asserted.

$$\text{WordEnable}[N] = \text{StartEnable}[N] \ \& \ \sim\text{StopEnable}[N]$$

So the whole mess takes only about 170 gates (including a few used as buffers). Of course we have to generate the value for the stop index, so for get and put operations we also need a 5-bit adder plus some additional gates to deal with objects that overflow to the next block.

Decoding the Boolean group of CPU instructions

The flag bits for the Boolean CPU instructions are encoded as follows:

001: and ($S_n \cdot D_n \rightarrow D_n$)

010: copy inverted ($\sim S_n \rightarrow D_n$)

011: and inverted ($\sim S_n \cdot D_n \rightarrow D_n$)

101: copy ($S_n \rightarrow D_n$)

110: or ($S_n + D_n \rightarrow D_n$)

111: xor ($(\sim S_n \cdot D_n) + (S_n \cdot \sim D_n) \rightarrow D_n$)

These encodings are chosen to make instruction decode simple. A typical logic unit is controlled by 4 control lines C0 through C3. Each control line enables one line from the truth table which maps the inputs S and D to the output as follows:

$$(C3 \cdot S_n \cdot D_n) + (C2 \cdot S_n \cdot \sim D_n) + (C1 \cdot \sim S_n \cdot D_n) + (C0 \cdot \sim S_n \cdot \sim D_n) \rightarrow O_n$$

With the chosen flag mappings, C0 through C3 can be generated as follows:

$$\sim F0 \cdot \sim (F2 \cdot F1) \rightarrow C0$$

$$F2 \rightarrow C2$$

$$F1 \rightarrow C1$$

$$(\sim F0 \cdot F1 \cdot F2) + (F0 \cdot \sim F1 \cdot F2) + (\sim F1 \cdot \sim F2) \rightarrow C3$$

The last equation has not been simplified but even so shows that the decode can be done with only two gate delays (given that the flag bits are available in both inverted and non-inverted form). These gate delays will cost nothing because they are parallel to the source operand routing delay, which is at least two gate delays.